Санкт-Петербургский государственный университет

Технологии программирования

Группа 20.Б09–мм

Вивдич Софья Андреевна

Модификация Java UI для удовлетворения требований проекта OpenJDK CRaC

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
старший преподаватель кафедры системного программирования СПбГУ
Козлов А.П.

Санкт-Петербург
2023

Оглавление

Введение **3**

1. Постановка задачи **5**

2. Обзор **6**

2.1. Базовая графическая библиотека AWT 6

2.2. X Window System 7

2.3. Сетевой протокол X11 8

2.4. Использование AWT другими графическими библиотеками 9

2.5. Наработки 10

3. Ход работы **11**

3.1. Описание AWT–приложения 11

3.2. Общение AWT–приложения с X Server 12

3.3. Описание проблемы 12

3.4. Поиск причины ошибок 13

3.5. Устранение проблемы 15

3.5.1. XWindow, XBaseWindow 15

3.5.2. AwtGraphicsConfigData, XVisualInfo 16

3.5.3. X11GraphicsConfig, XRGraphicsConfig 18

3.5.4. X11GraphicsDevice, X11GraphicsEnvironment . . . 20

3.5.5. Реализация решения 21

4. Проверка результата **23**

Заключение **25**

Список литературы **26**

Введение

Согласно индексу TIOBE, на протяжении двадцати лет Java входит в тройку самых популярных языков программирования в мире [16]. Это связано с кроссплатформенностью, надежностью, поддержкой высокоуровнего параллелизма, обширной библиотекой классов, которая позволяет писать программы для самых разных целей. Java активно используется в веб–разработке, мобильной разработке, науке, программных инструментах, в анализе данных.

Для исполнения Java–приложений необходима JRE (Java Runtime Environment) — это минимальная реализация JVM (Java Virtual Machine, исполняет байт-код), включающая в себя стандартный набор библиотек классов. JRE и набор инструментов для разработки приложений в совокупности образуют JDK (Java Development Kit). Эталонной реализацией Java считается OpenJDK.

В настоящее время Java является одним из самых быстрых языков по времени выполнения, немного уступая популярным C, C++ [11]. Однако для Java–программ характерны медленный старт и необходимость предварительного разогрева. Это связано с загрузкой и инициализацией множества классов, из которых состоит запускаемая Java–программа. Чтобы снизить влияние загрузки классов на скорость старта программы, применяются такие технологии как Application Class Data Sharing [2], динамическая JIT–компиляция, Ahead–of–Time компиляция. Однако эти процессы также требуют немало времени, что не всегда позволяет увидеть ощутимое преимущество от их использования.

Одним из решений проблемы медленного старта Java–программ является проект CRaC (Coordinated Restore at Checkpoint), созданный на основе OpenJDK [17], [18]. Применение CRaC позволяет единожды разогреть виртуальную машину, сохранить ее состояние в образ при помощи операции checkpoint и затем в любой необходимый момент времени восстановить состояние из сохраненного образа, то есть запустить уже разогретую виртуальную машину. Восстановление состояния обес-

печивается операцией restore.

CRaC гарантирует, что восстановленный процесс будет полностью совпадать с исходным. Если CRaC выявляет наличие опасных состояний и предвидит проблемы с восстановлением, то отменяет checkpoint. Такие проблемы может вызвать связь приложения с внешней средой, например, с графической подсистемой. Это говорит о том, что для любого приложения, так или иначе связанного с графикой, невозможно создание образа и, соответственно, применение CRaC.

Поддержка графики в проекте CRaC решит проблему медленного старта уже существующих программ и сделает использование Java для разработки графических приложений более привлекательным.

В развитии применения CRaC к графическим приложениям уже есть некоторые результаты: удается имитировать восстановление простых Swing–приложений, выполнена часть работы по восстановлению AWT–приложений.

1. Постановка задачи

Целью данной работы является поддержка графических приложений в проекте OpenJDK CRaC.

Для достижения цели были поставлены следующие задачи.

1. Сделать обзор библиотеки AWT для разработки графических приложений.
2. Определить взаимосвязь AWT с другими графическими библиотеками.
3. Устранить проблемы при восстановлении простого AWT–приложения.

2. Обзор

В данном разделе представлены обзор базовой графической библиотеки AWT, описание связи AWT с более современными и популярными графическими библиотеками, наработки по поддержке AWT проектом CRaC.

2.1. Базовая графическая библиотека AWT

AWT (Abstract Window Toolkit) — это первая библиотека Java для создания графических приложений на основе окон, часть стандартного API для реализации графического интерфейса в Java [1].

AWT предоставляет набор компонентов, из которых строится графическое приложение. Каждый компонент (кнопки, списки, текстовые поля и др.) представляет собой отдельный класс, расширяющий абстрактный класс Component, который описывает некоторые свойства всех графических компонентов, например, положение, цвет, шрифт.

Создание приложения невозможно без контейнеров — компонентов, в которые помещаются другие компоненты. Контейнеры описываются классом Container, который расширяют классы Window и Panel. При разработке графического приложения в первую очередь необходимо создать окно верхнего уровня — Frame или Dialog, наследники Window. Пользователь может установить размер и заголовок окна. Все остальные компоненты помещаются в контейнер Panel, который в свою очередь содержится в окне верхнего уровня. Иерархия классов AWT приведена на картинке 1.

Все компоненты AWT имеют соответствующую нативную структуру. Это означает, что AWT поручает отрисовку и другие действия над графическим объектом соответствующему компоненту графической системы, который называется peer, для которого реализованы все функциональные возможности. Такой подход обеспечивает переносимость компонентов, однако приводит к разному внешнему виду одного и того же приложения на разных платформах.

Библиотека AWT предоставляет возможность обрабатывать собы-

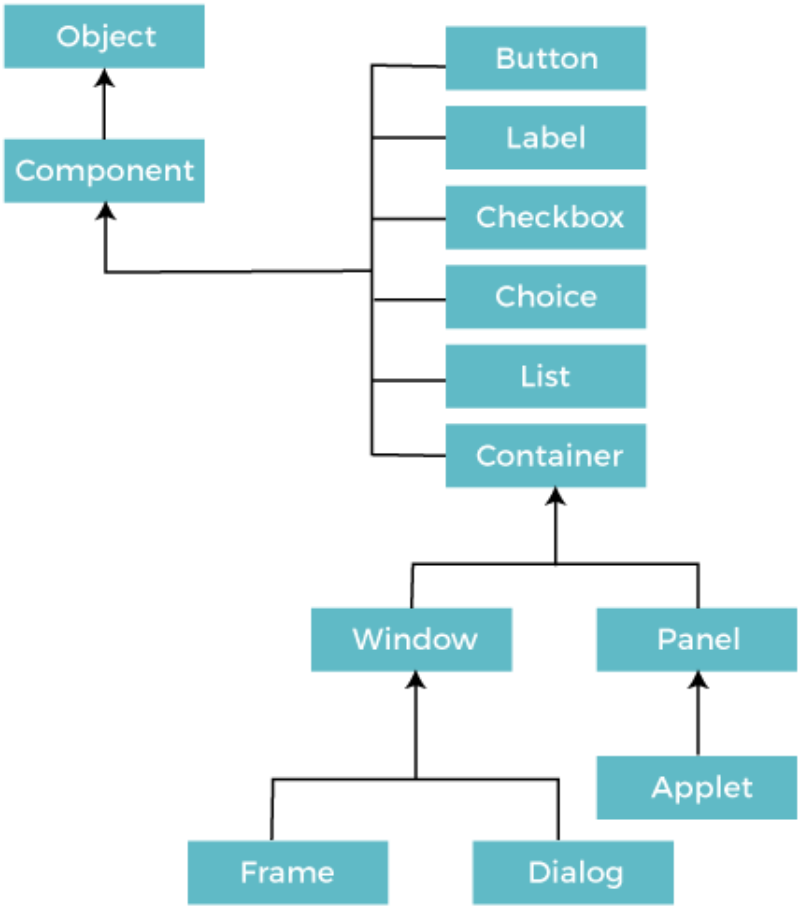

Рис. 1: Иерархия классов AWT

тия, вызванные действиями пользователя над какими-либо объектами, например, при нажатии на кнопку или ввод данных в текстовое поле. Объект, в результате изменения состояния которого сгенерировалось событие, называется ”источник события”. При разработке приложения для каждого потенциального источника события регистрируют слушателя — объект, который должным образом обрабатывает сгенерированное событие.

2.2. X Window System

X Window System — это оконная клиент–серверная система, которая предоставляет инструменты и протоколы для создания графического пользовательского интерфейса в UNIX–подобных системах [12].

X Window System состоит из трех частей:

- X Server — управляет дисплеем и оборудованием ввода. Захватывая данные с устройств ввода (клавиатура, мышь), X Server генерирует события в ответ на действия пользователя и отправляет их программе на обработку. Другая важная функция X Server заключается в получении и обработке запросов от графического приложения, направленных на выполнение каких-либо действий над окнами. Под контролем менеджера окон X Server отображает результат выполнения запроса, например, обновляет соответствующее окно.

- Windows Manager — менеджер окон — приложение, основная функция которого заключается в управлении графическими свойствами дисплея и клиентскими окнами, изменении их размера и положения [13].

- X Client — графическое приложение, которое взаимодействует с X Server. Клиент отправляет серверу запросы, которые содержат в себе действие (например, создание окна или изменение некоторых свойств) и идентификаторы ресурсов, над которыми запрашивается выполнение действия.

2.3. Сетевой протокол X11

X11 — сетевой графический протокол, который обеспечивает соединение между X Client и X Server. Клиентская часть X11 представляет собой библиотеку Xlib, которую клиенты (графические приложения) используют для взаимодействия с оконной системой [7]. Xlib реализована на C. Общение графического приложения и сервера изображено на рисунке 2.

На данной схеме изображено графическое приложение User–Space Program, которое использует компоненты инструментария для работы с графикой. Инструментарий связан с рассмотренной библиотекой Xlib, которая в свою очередь по протоколу X11 связана с X Server. Таким

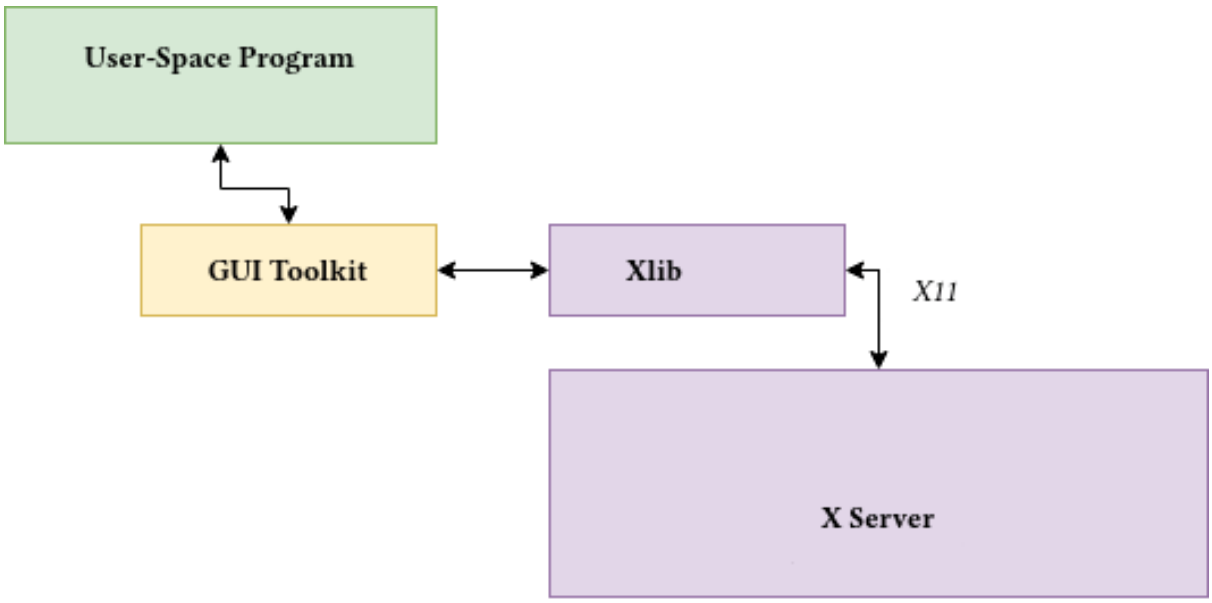

Рис. 2: Общение X client и X server

образом, протокол X11 отвечает за доставку сообщений между графическим приложением и сервером.

Как уже отмечалось ранее, библиотека Xlib написана на C. Подключение библиотеки к инструментарию AWT возможно благодаря механизму JNI(Java Native Interface), который позволяет вызывать функции на C из Java–программы.

Таким образом, библиотека AWT использует протокол X11 для работы с нативными компонентами операционной системы.

2.4. Использование AWT другими графическими библиотеками

AWT — исторически первая библиотека для работы с графикой на Java.

Следующая библиотека Swing полностью написана на Java и не делегирует действия нативным компонентам графической подсистемы, поэтому ее компоненты являются легковесными. В отличие от AWT,

библиотека Swing минимально использует графическую подсистему, а именно ей необходимо только окно операционной системы, на котором будет отрисовка всех остальных компонентов.

JavaFX — инструментарий, позволяющий работать с двумерной и трехмерной графикой. JavaFX предоставляет возможности для создания сложных интересных графических приложений для удовлетворения требований любой задачи. Одной из особенностей JavaFX является возможность взаимодействия с библиотекой Swing, а значит и с AWT.

Почти все графические Java–приложения используют описанные инструментарии.

Для запросов, которые касаются графики операционной системы, все инструментарии используют протокол X11 с помощью AWT, а значит поддержка AWT проектом CRaC станет серьезным прорывом в поддержке и других графических библиотек. В частности, это позволит задуматься о применении технологии CRaC к IntelliJ IDEA, как к большому Swing–приложению.

2.5. Наработки

Работа по поддержке AWT проектом CRaC началась с создания простого AWT–приложения. При изучении общения приложения с сервером были найдены и исправлены многие ошибки, из-за которых приложение не могло восстановиться [8]. В основном ошибки были связаны с очисткой не всех данных после checkpoint, то есть оставались связи сохраненного в образ и затем физически уничтоженного приложения с графической подсистемой. На момент начала работы простое приложение, состоящие только из окна, не всегда восстанавливается после checkpoint, при неуспешных запусках наблюдаются ошибки.

3. Ход работы

3.1. Описание AWT–приложения

Работа проводится над следующей программой. Сначала создается Frame — окно верхнего уровня, необходимая составляющая любого AWT–приложения. Окну устанавливается размер, название "AWTApp", видимость. Запущенное приложение можно увидеть на рисунке 3.

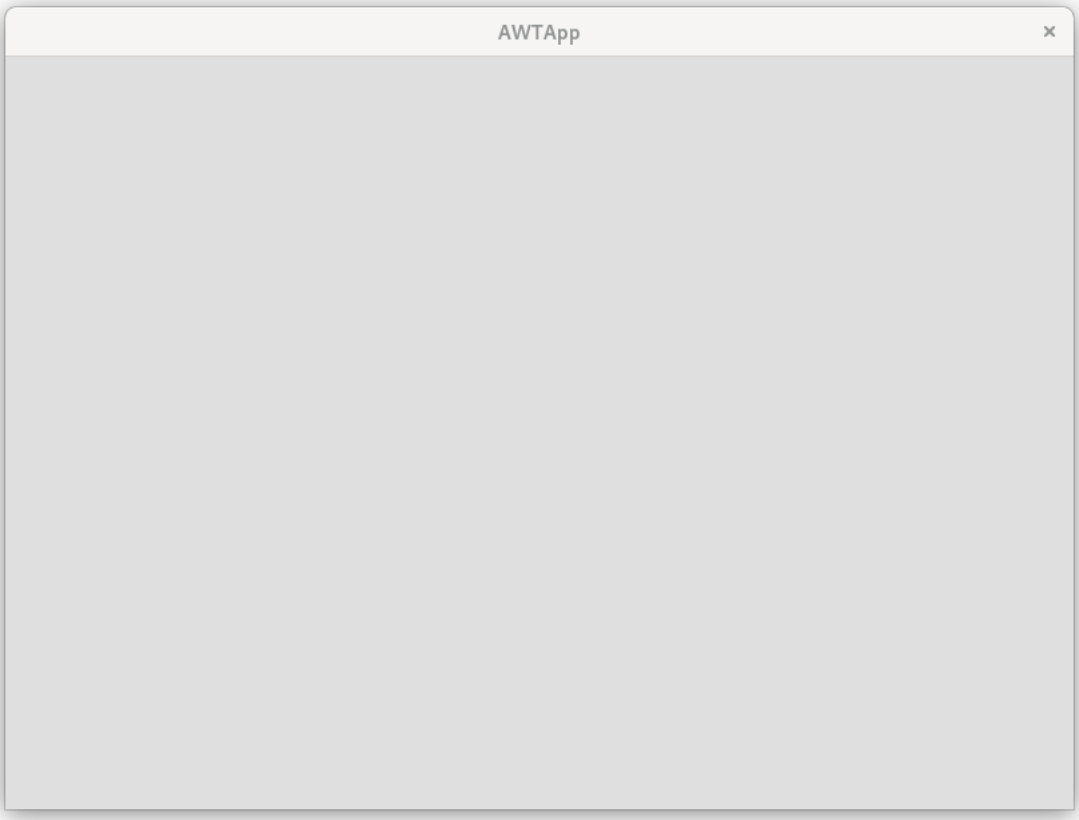

Рис. 3: Простое AWT–приложение, состоящее из Frame

После создания окна вызывается checkpoint для сохранения образа и последующего восстановления. При checkpoint связь приложения с графической подсистемой разрывается, а именно созданный раннее Frame, объект Java, теряет связь с соответствующим ему нативным компонентом графической подсистемы. Затем при выполнении операции restore, то есть при восстановлении, происходит попытка связать исходный Frame с новым нативным компонентом, которому будет делегироваться отрисовка. Ожидается, что после restore окно восстановится

и будет выглядеть в точности как на рисунке 3, однако восстановление окна происходит не всегда.

3.2. Общение AWT–приложения с X Server

Общение AWT–приложения (X Client) с X Server изучается с помощью инструмента xscope. Xscope находится между клиентом и сервером и распечатывает содержимое каждого запроса, ответа, ошибки или события, которые передаются между ними [15]. Стоит отметить, что в данном случае с сервером по очереди общаются два клиента: окно приложения до checkpoint — первое взаимодействие с сервером — первый клиент, окно после restore — второй.

3.3. Описание проблемы

При помощи xscope были сохранены логи общения клиентов с сервером. В ходе взаимодействия первого клиента с сервером нет никаких проблем: клиент здесь — обычное AWT–приложение, окно, которое отображается на дисплее.

Со вторым клиентом, восстанавливающимся при restore окном, есть проблемы: на запрос клиента CreateWindow [4], создание окна приложения, сервер отвечает ошибкой Match [5]. Ниже приведен соответствующий фрагмент логов (для упрощения понимания незначимая часть запроса вырезана).

```
1.37: Client 2 (pid 157474 java) -->  276 bytes
   ............REQUEST: CreateWindow
                 depth: 18
                   wid: WIN 03200007
                parent: WIN 000006aa
                     x: 0
                     y: 32
                 width: 0316
                height: 023a
          border-width: 0000
                 class: InputOutput
                visual: VIS e8529008
 1.37:           192 bytes <-- X11 Server 2 (pid 1257 Xorg)
```

```
..............ERROR: Match
               minor opcode: 0000
               major opcode: 01
```

На все последующие запросы сервер отвечает ошибками BadGlyphSet и Drawable.

3.4. Поиск причины ошибок

Чтобы отследить, какие действия при восстановлении приложения приводят к ошибкам, в первую очередь сравнивались логи общения первого клиента с сервером с логами общения второго клиента с сервером. Так как необходимо восстанавление приложения в том же виде, в каком оно было сохранено, ожидалось, что логи общения клиентов с сервером будут совпадать. Однако применение CRaC все же вносит некоторые изменения, поэтому логи иногда могут отличаются. Необходимо выяснить, какие отличия допустимы, а какие влекут за собой ошибки.

При многократном запуске тестового приложения было обнаружено, что иногда окно все–таки восстанавливается, причем без какой–либо закономерности. Было принято решение запускать программу до тех пор, пока окно не восстановится, сохранить логи общения этого клиента (восстановленное окно) с сервером и сравнить их с логами, в которых наблюдаются ошибки. Логи восстановленного окна были сохранены. Сервер не отвечает на запрос CreateWindow при успешном создании окна, поэтому ниже приведен ответ сервера на дополнительный запрос ChangeProperty. Как можно заметить, никаких ошибок нет.

```
1.27: Client 2 (pid 533188 java) -->  276 bytes
    ............REQUEST: CreateWindow
                 depth: 18
                   wid: WIN 03600007
                parent: WIN 000006aa
                     x: 0
                     y: 32
                 width: 0316
                height: 023a
```

```
            border-width: 0000
                   class: InputOutput
                  visual: VIS 00000021
    ............REQUEST: ChangeProperty
                    mode: Replace
                  window: WIN 03600007
                property: <WM_NAME>
                    type: <STRING>
                  format: 08
                    data: ''AWTApp''
 1.27:                 160 bytes <-- X11 Server 2 (pid 1260 Xorg)
                ..............EVENT: PropertyNotify
                              window: WIN 03600007
                                atom: <WM_NAME>
                                time: TIM 07f493a7
                               state: NewValue
```

До запроса CreateWindow логи восстановленного окна полностью совпадают с логами не восстановленного окна. В самом запросе CreateWindow у логов отличается только одна строка — значение visual.

Visual — это структура, которая описывает доступную визуальную конфигурацию экрана [9]. При создании окна в запросе CreateWindow значение visual можно указать через VISUALID или CopyFromParent. Во втором случае тип визуального элемента берется из родительского окна.

Все окна, принадлежащие приложениям, разделают между собой главное корневое окно — дисплей. Он покрывает все экраны, подключенные к серверу. Дисплей в приведенных запросах выглядит как "parent: WIN 000006aa". Значение visual у корневого окна: VIS 00000021.

В обычном AWT–приложении до checkpoint и в случае окна, которое удалось восстановить, при формировании запроса на создание окна значение visual "VIS 00000021" совпадает с родительским окном — дисплеем, за счет чего оба окна корректно отображаются на экране. В логах не восстановленного окна значение visual "VIS e8529008". Окно приложения относится к классу InputOutput, поэтому его визуальный тип, не поддерживаемый экраном (как в этом случае), согласно документации, приводит к ошибке Match [4].

Таким образом, была обнаружена проблема, из–за которой попытка

восстановления окна заканчивается ошибками. Все запросы к X Server, в том числе на создание окна, формируются в ранее упомянутой библиотеке Xlib. С помощью отладчика gdb было установлено, что в нативный метод формирования запроса на создание окна приходит уже неверное значение visual. Это означает, что некорректный visual формируется на стороне Java.

3.5. Устранение проблемы

Для обеспечения корректного формирования параметра visual и, как следствие, устранения ошибки при восстановлении окна был исследован и модифицирован ряд AWT–классов, которые на разных уровнях задействованы в создании и отображении окна на экране.

3.5.1. XWindow, XBaseWindow

Для создания окон AWT использует классы X11, образующие глубокую иерархию наследования. В первую очередь, был рассмотрен класс **XBaseWindow** [23], находящийся на вершине иерархии, так как из этого класса вызывается нативный метод создания окна. В метод *create* этого класса передаются параметры, по которым необходимо создать окно. В ходе выполнения этого метода параметры проверяются и обогащаются, после чего передаются в нативную функцию XCreateWindow [30] библиотеки Xlib для непосредственно создания физического окна. С помощью отладочной печати выяснилось, что параметр visual инициализирован до метода *create*. Кроме того, при исследовании действий над параметрами в рамках метода было установлено, что они не затрагивают параметр visual и не могут изменить его значение на некоррет-ное.

Инициализация параметров происходит в методе *preInit*, который вызывается перед *create*. Большая часть характеристик для создания окна заполняется в *preInit* класса **XWindow** [25], потомок рассмотренного класса XBaseWindow. В переопределенном методе инициализируются основные параметры, один из которых — visual.

Метод *preInit* начинается с вызова *initGraphicsConfiguration*, в котором инициализируется два объекта:

- экземпляр класса **X11GraphicsConfig** [20] — графическая конфигурация экрана на основе значения одноименного поля объекта target. Target — это экземпляр класса Component [3], который представляет собой целевой объект, в данном случае это окно Frame;

- экземпляр **AwtGraphicsConfigData** — конструируется на основе данных объекта X11GraphicsConfig из предыдущего пункта. Данные представляют собой адрес в памяти, по которому с помощью смещения можно получать значения полей.

После выполнения *initGraphicsConfiguration* в методе *preInit* в параметры добавляется характеристика visual, которая инициализируется следующим образом:

- у объекта AwtGraphicsConfigData определяется значение поля типа **XVisualInfo** [14];

- из полученного экземпляра класса XVisualInfo, который устроен аналогично AwtGraphicsConfigData, таким же образом с помощью смещения относительно начала данных определяется значение visual.

Для отслеживания значения visual на нативном уровне были созданы вспомогательные нативные методы в библиотеке Xlib [26]. С их помощью и отладочной печати было установлено, что значение visual некорректно уже на этом этапе, начиная с инициализации параметра. В связи с этим было решено изучить упомянутые выше классы, которые затрагивают формирование значения характеристики visual.

3.5.2. AwtGraphicsConfigData, XVisualInfo

Классы **AwtGraphicsConfigData** и **XVisualInfo** рассматриваются вместе, так как генерируется на основе одного и того же класса

WrapperGenerator [19]. У обоих классов есть опорное поле, значение которого — адрес в памяти. Данные о связанных с классом объектах и значения характеристик можно получить засчет смещения адреса. Так, в классе AwtGraphicsConfigData есть метод, позволяющий получить соответствующий конфигурации объект XVisualInfo.

Класс XVisualInfo используется для хранения и получения информации о визуальных характеристиках, которые поддерживаются экраном. Одна из них — проблемный параметр visual. Получить значение этой характеристики в классе XVisualInfo можно так же с помощью смещения относительно опорного адреса. Именно таким образом инициализируется visual в рассмотренном ранее методе *preInit* класса **XWindow**.

Для того чтобы отследить проблемный параметр в куче, где хранятся все объекты, созданные для нужд Java–программы, было принято решение сделать дамп кучи — снимок памяти в данный момент времени. Для этих целей был использован инструмент VisualVM [10]. Куча просматривалась и изучалась в три этапа:

- состояние программы до checkpoint, обычное AWT–приложение;
- состояние в момент выполнения операции checkpoint. Снимок кучи на этом этапе проводился с целью выявления объектов, которые не были очищены до checkpoint;
- состояние после операции restore — окно либо восстановилось, либо нет;

Исследование дампа кучи осложнялось тем, что объекты, которые необходимо было рассмотреть, в описанных классах не хранятся как поля, их значения можно получить только с помощью вызова соответствующих методов. Для решения этой проблемы был модифицирован класс генератора WrapperGenerator [19] таким образом, чтобы все необходимые данные, связанные с классом, хранились как поля. Кроме того, были добавлены вспомогательные поля для упрощения отслеживания объектов классов в куче.

Все описанные действия позволили установить, что опорное поле, которое хранит адрес в памяти, объектов классов AwtGraphicsConfigData и XVisualInfo до операции checkpoint и после операции restore имеет одно и то же значение, то есть при восстановлении окна работа ведется с тем же участком памяти, что и при создании окна оригинального приложения. При восстановлении приложения используемые ранее адреса неактуальны и некорректны, так как память перевыделяется. Значит, опорный адрес этих объектов, используемый при запуске оригинального приложения, где–то сохраняется и переиспользуется при восстановлении. Иногда это приводит к восстановлению окна, но чаще — к неудаче, потому что в повторно используемом участке памяти хранится не то, что ожидается приложением. Исследование кучи, отладочная печать и добавление вспомогательных полей показали, что адрес кэшируется не в объектах рассмотренных классов. Опорный адрес передается в конструктор класса AwtGraphicsConfigData в методе *initGraphicsConfiguration* класса XWindow от одноименного поля класса X11GraphicsConfig.

3.5.3. X11GraphicsConfig, XRGraphicsConfig

Следующим на рассмотрении был класс **X11GraphicsConfig** [20], который описывает характеристики графического устройства, такого как принтер или монитор [6]. Этот класс содержит поле, которое хранит адрес в памяти. Значение этого поля передается в конструктор рассмотренного выше класса AwtGraphicsConfigData.

В конструктор X11GraphicsConfig была добавлена отладочная печать, с помощью которой выяснилось, что за все время работы программы конструктор вызывается один раз, а значит, создается только один экземпляр X11GraphicsConfig. В связи с этим появилось предположение о том, что адрес кэшируется в объекте класса X11GraphicsConfig, так как при восстановлении приложения объект X11GraphicsConfig используется (конфигурация хранится как поле у target — целевой объект, окно Frame), но не пересоздается. Чтобы убедиться в этом, в момент checkpoint был рассмотрен дамп кучи. Ожидалось, что в куче

будет наблюдаться объект X11GraphicsConfig, который хранит данные о графической конфигурации с момента запуска оригинального приложения. Однако в куче нет экземпляров класса X11GraphicsConfig, как можно видеть на рисунке 4, их количество равно нулю.

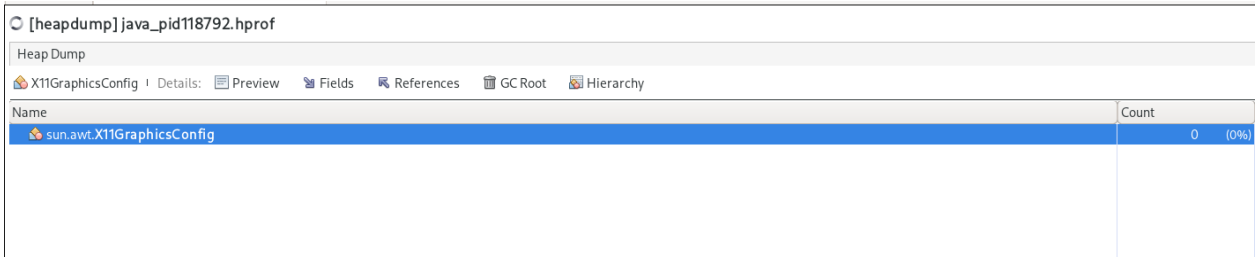

Рис. 4: Heap Dump во время выполнения операции checkpoint. Экземпляры класса X11GraphicsConfig

При дальнейших исследованиях дампа кучи во время выполнения операции checkpoint было обнаружено, что в куче хранится экземпляр класса **XRGraphicsConfig** [24] (рис. 5) — потомок X11GraphicsConfig. Кроме того, было установлено, что графическая конфигурация объекта target — это экземпляр XRGraphicsConfig. Конструктор этого класса, как и конструктор суперкласса X11GraphicsConfig, вызывается за всю программу один раз, а значит, экземпляр XRGraphicsConfig создается при запуске оригинального окна и повторно используется при восстановлении.

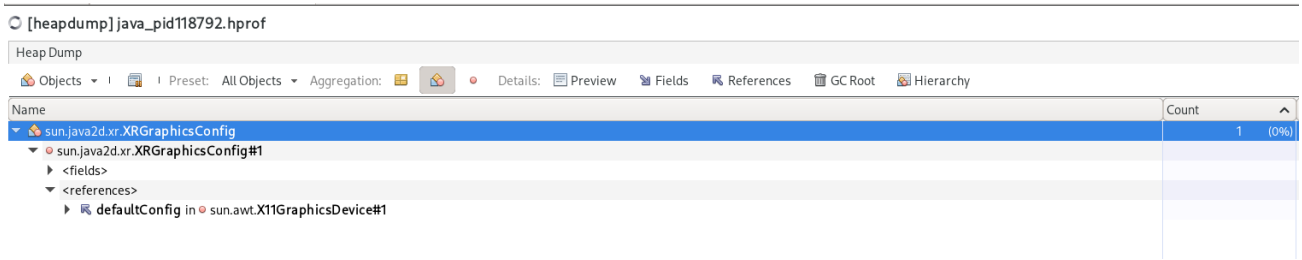

Рис. 5: Heap Dump во время выполнения операции checkpoint. Экземпляры класса XRGraphicsConfig

Таким образом, был найден объект, который кэширует данные о графической конфигурации с запуска оригинального приложения, не очищается перед checkpoint и не реинициализируется после, из–за чего при восстановлении окна используется неактуальная конфигурация.

Объект XRGraphicsConfig закреплен за **X11GraphicsDevice** [21] как его поле defaultConfig (рис. 5).

3.5.4. X11GraphicsDevice, X11GraphicsEnvironment

Класс **X11GraphicsEnvironment** [22] представляет собой реализацию графического окружения для платформы UNIX. Это singletone–объект, который на стороне AWT отвечает за работу диспея. С объектом класса X11GraphicsEnvironment связаны доступные экраны X11, представленные классом **X11GraphicsDevice** [21].

Класс X11GraphicsDevice описывает графические устройства (в том числе экран), которые могут быть доступны в определенной графической среде. Каждое устройство имеет один или несколько связанных с ним объектов графической конфигурации, которые определяют, в каких конфигурациях может быть использовано устройство. Конфигурация по умолчанию хранится в отдельном поле defaultConfig.

Графическая конфигурация экрана инициализируется до создания окна приложения. Происходит это следующим образом:

- у объекта X11GraphicsEnvironment извлекается доступный экран по умолчанию;

```
Window initGraphicsConfiguration
X11GraphicsEnvironment getDefaultScreenDevice()
```

- у экрана рассматривается поле defaultConfig:

```
X11GraphicsDevice getDefaultConfiguration()
```

  - если значение равно null, вызывается метод инициализации графической конфигурации на основе текущего запуска;

```
X11GraphicsDevice getDefaultConfiguration() case defaultConfig ==
    NULL
X11GraphicsDevice initialize defaultConfig
```

  - иначе значение поля не меняется;

- значение defaultConfig становится конфигурацией целевого окна target.

```
Window.setGraphicsConfiguration(): new GC is
+ X11GraphicsConfig[dev=X11GraphicsDevice[screen=0],vis=0x21]
+ this is java.awt.Frame[frame0,0,0,0x0,invalid,hidden,normal]
```

При первом запуске приложения (оригинальное окно) defaultConfig корректно инициализируется, однако перед вторым запуском (попытка восстановления окна) поле не очищается и, в соответствии с алгоритмом выше, не проходит реинициализацию, как можно видеть в логах ниже:

```
Try to make a checkpoint and then restore...
Checkpoint restored

Window initGraphicsConfiguration
X11GraphicsEnvironment getDefaultScreenDevice()
X11GraphicsDevice getDefaultConfiguration()
Window.setGraphicsConfiguration(): new GC is
+ X11GraphicsConfig[dev=X11GraphicsDevice[screen=0],vis=0x21]
+ this is java.awt.Frame[frame1,0,0,0x0,invalid,hidden,normal]
```

В результате, все дальнейшие действия выполняются с неактуальной конфигурацией, которая не поддерживается текущим запуском.

3.5.5. Реализация решения

Для устранения описанной проблемы в класс X11GraphicsDevice был добавлен метод *beforeCheckpoint* [28], в котором происходит очистка поля defaultConfig. Метод вызывается из одноименного метода класса X11GraphicsEnvironment [29] перед операцией checkpoint. Такая реализация обеспечивает очистку данных о конфигурации предыдущего запуска. При выполнении операции restore, как и в случае с запуском обычного AWT–приложения, до создания окна происходит инициализация графической конфигурации. В соответствии с приведенным выше алгоритмом, у объекта X11GraphicsEnvironment извлекается экран по умолчанию, у которого рассматривается значение поля defaultConfig. Вызов метода *beforeCheckpoint* сбросил значение после первого запуска, в результате defaultConfig равен null, значит, будет вызван метод инициализации графической конфигурации. Таким образом, при восстановлении приложения defaultConfig реинициализируется корректным

значением для текущего запуска.

Стоит отметить, что приведенный алгоритм выполняется при запуске любого привычного AWT–приложения, использование CRaC не добавляет никакой новой логики в процесс инициализации графической конфигурации.

4. Проверка результата

Для проверки результата были выполнены следующие действия:

- повторный дамп кучи в момент выполнения checkpoint (рис. 6, 7) показал отсутствие экземпляров классов, связанных с графической конфигурацией, что означает, что данные о прошлом запуске больше не кэшируются и не могут быть переиспользованы;

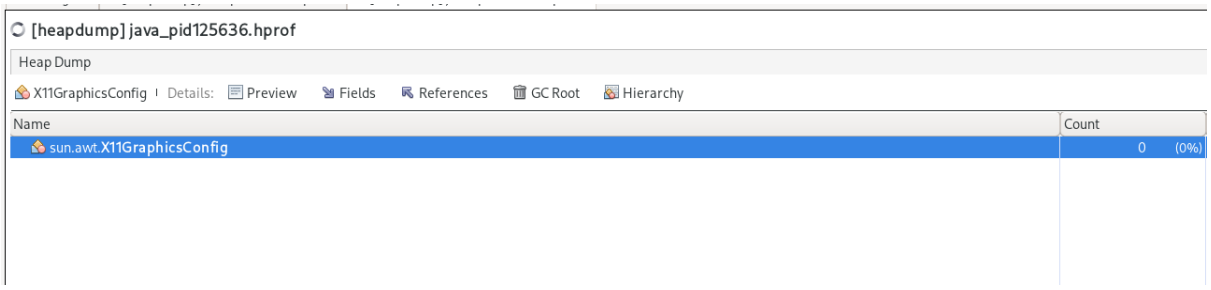

Рис. 6: Heap Dump во время выполнения операции checkpoint. Запуск после устранения проблемы. Экземпляры класса X11GraphicsConfig

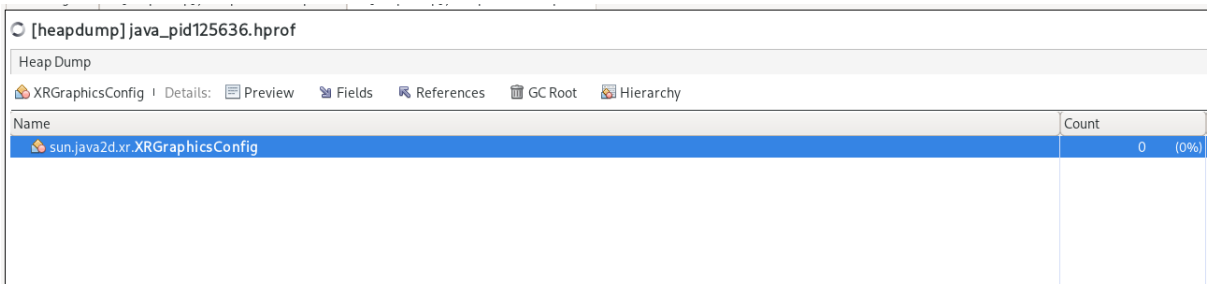

Рис. 7: Heap Dump во время выполнения операции checkpoint. Запуск после устранения проблемы. Экземпляры класса XRGraphicsConfig

- с помощью отладочной печати было установлено, что инициализация конфигурации при восстановлении приложения проходит те же самые этапы, что и при оригинальном запуске, а значит, конфигурация соответствует запуску. Логи оригинального запуска:

```
Window initGraphicsConfiguration
X11GraphicsEnvironment getDefaultScreenDevice()
X11GraphicsDevice getDefaultConfiguration()
X11GraphicsDevice getDefaultConfiguration() case defaultConfig == NULL
X11GraphicsDevice initialize defaultConfig
```

```
Window.setGraphicsConfiguration(): new GC is
+ X11GraphicsConfig[dev=X11GraphicsDevice[screen=0],vis=0x21]
+ this is java.awt.Frame[frame0,0,0,0x0,invalid,hidden,normal]
```

Логи восстановления:

```
Try to make a checkpoint and then restore...
Checkpoint restored

Window initGraphicsConfiguration
X11GraphicsEnvironment getDefaultScreenDevice()
X11GraphicsDevice getDefaultConfiguration()
X11GraphicsDevice getDefaultConfiguration() case defaultConfig == NULL
X11GraphicsDevice initialize defaultConfig
Window.setGraphicsConfiguration(): new GC is
+ X11GraphicsConfig[dev=X11GraphicsDevice[screen=0],vis=0x21]
+ this is java.awt.Frame[frame1,0,0,0x0,invalid,hidden,normal]
```

- с использованием нативных методов [26] было проверено, что параметр visual, который является частью графической конфигурации, принимает корректное значение, поддерживаемое экраном. Ошибки, обнаруженные ранее с помощью xscope, пропали, окно корректно восстанавливается.

Заключение

В ходе выполнения данной работы были достигнуты следующие результаты:

- сделан обзор библиотеки AWT для разработки графических приложений;
- определена взаимосвязь AWT с другими графическими библиотеками;
- выявлена и устранена проблема, из-за которой AWT–приложение не восстанавливалось. На данный момент окно корректно восстанавливается, тем самым завершена реализация начальной поддержки AWT–приложений проектом CRaC. Для достижения результата были модифицированы классы:
  - WrapperGenerator, Window, XVisualInfo, Component, XWindow, AwtGraphicsConfigData, XBaseWindow, XlibWrapper.c (добавлены нативные методы), X11GraphicsConfig, XRGraphicsConfig — для выявления проблемы;
  - X11GraphicsDevice, X11GraphicsEnvironment — реализация решения, устранение проблемы.

Код доступен по ссылке [27].

Список литературы

[1] Abstract Window Toolkit. — Access mode: https://ru.wikipedia.org/wiki/Abstract_Window_Toolkit (online; accessed: 2023-05-15).

[2] Application Class Data Sharing. — Access mode: https://medium.com/@toparvion/appcds-for-spring-boot-applications-first-contact-6216db6a4194 (online; accessed: 2023-05-15).

[3] Class Component documentation. — Access mode: https://docs.oracle.com/javase/8/docs/api/java/awt/Component.html (online; accessed: 2023-05-14).

[4] CreateWindow Request. — Access mode: https://www.x.org/releases/X11R7.7/doc/xproto/x11protocol.html#requests:CreateWindow (online; accessed: 2023-05-15).

[5] Error Match. — Access mode: https://tronche.com/gui/x/xlib/event-handling/protocol-errors/default-handlers.html (online; accessed: 2023-05-15).

[6] GraphicsConfiguration documentation. — Access mode: https://docs.oracle.com/javase/8/docs/api/java/awt/GraphicsConfiguration.html (online; accessed: 2023-05-14).

[7] Introduction to Xlib. — Access mode: https://tronche.com/gui/x/xlib/introduction/ (online; accessed: 2023-05-15).

[8] Results of previous work CRaC AWT. — Access mode: https://github.com/AntonKozlov/cracui/pull/13 (online; accessed: 2023-05-15).

[9] Visual Description. — Access mode: https://metacpan.org/pod/X11::Xlib::Visual (online; accessed: 2023-05-15).

[10] VisualVM documentation. — Access mode: https://visualvm.github.io/documentation.html (online; accessed: 2023-05-14).

[11] What Are the Greenest Programming Languages? Time: Execution time ranking. — Access mode: https://medium.com/codex/what-are-the-greenest-programming-languages-e738774b1957 (online; accessed: 2023-05-15).

[12] X Window System: Definition. — Access mode: https://dic.academic.ru/dic.nsf/ruwiki/12682 (online; accessed: 2023-05-15).

[13] X Window System: Windows manager. — Access mode: https://www.techopedia.com/definition/10101/x-window-system (online; accessed: 2023-05-15).

[14] XVisualInfo documentation. — Access mode: http://www.asvcorp.ru/tech/linux/xwinprg/structs/XVisualInfo.html (online; accessed: 2023-05-14).

[15] Xscope - X Window Protocol Viewer. — Access mode: https://www.x.org/releases/X11R7.6/doc/man/man1/xscope.1.xhtml (online; accessed: 2023-05-15).

[16] Индекс TIOBE самых популярных языков программирования. — Access mode: https://www.tiobe.com/tiobe-index/ (online; accessed: 2023-05-15).

[17] Проект OpenJDK CRaC. — Access mode: https://openjdk.java.net/projects/crac/ (online; accessed: 2023-05-15).

[18] Реализация CRaC. — Access mode: https://github.com/openjdk/crac (online; accessed: 2023-05-15).

[19] Реализация класса WrapperGenerator. — Access mode: https://github.com/SVivdich02/cracui-1/blob/x11-dev/make/jdk/src/classes/build/tools/x11wrappergen/WrapperGenerator.java (online; accessed: 2023-05-14).

[20] Реализация класса X11GraphicsConfig. — Access mode: https://github.com/SVivdich02/cracui-1/blob/x11-dev/src/

java.desktop/unix/classes/sun/awt/X11GraphicsConfig.java (online; accessed: 2023-05-14).

[21] Реализация класса X11GraphicsDevice. — Access mode: https://github.com/SVivdich02/cracui-1/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11GraphicsDevice.java (online; accessed: 2023-05-14).

[22] Реализация класса X11GraphicsEnvironment. — Access mode: https://github.com/SVivdich02/cracui-1/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11GraphicsEnvironment.java (online; accessed: 2023-05-14).

[23] Реализация класса XBaseWindow. — Access mode: https://github.com/SVivdich02/cracui-1/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11/XBaseWindow.java (online; accessed: 2023-05-14).

[24] Реализация класса XRGraphicsConfig. — Access mode: https://github.com/SVivdich02/cracui-1/blob/x11-dev/src/java.desktop/unix/classes/sun/java2d/xr/XRGraphicsConfig.java (online; accessed: 2023-05-14).

[25] Реализация класса XWindow. — Access mode: https://github.com/SVivdich02/cracui-1/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11/XWindow.java (online; accessed: 2023-05-14).

[26] Реализация нативных методов в Xlib для проверки visual. — Access mode: https://github.com/SVivdich02/cracui-1/blob/x11-dev/src/java.desktop/unix/native/libawt_xawt/xawt/XlibWrapper.c#L317 (online; accessed: 2023-05-14).

[27] Репозиторий проекта cracui. Реализация решения. — Access mode: https://github.com/AntonKozlov/cracui/pull/14 (online; accessed: 2023-05-15).

[28] Устранение проблемы на стороне X11GraphicsDevice. Очистка конфигураций прошлого запуска. — Access mode: https://github.com/SVivdich02/cracui-1/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11GraphicsDevice.java#L77 (online; accessed: 2023-05-14).

[29] Устранение проблемы на стороне X11GraphicsEnvironment. — Access mode: https://github.com/SVivdich02/cracui-1/blob/x11-dev/src/java.desktop/unix/classes/sun/awt/X11GraphicsEnvironment.java#L63 (online; accessed: 2023-05-14).

[30] Функция XCreateWindow библиотеки Xlib. — Access mode: http://www.asvcorp.ru/tech/linux/xwinprg/functions/XCreateWindow.html (online; accessed: 2023-05-14).