## 一、动态改变状态值变量

在开发这款游戏中，我们遇到了一个挑战，涉及到如何在应用程序中动态更新游戏角色的状态（例如生命值和防御值）。这些属性需要根据玩家的行为实时更新，而这通常是通过事件监听器来实现的，例如按下增加或减少的按钮。

### 1.问题描述

Java 中的事件监听器（如 ActionListener），当涉及到在匿名内部类中访问和修改外部变量时，这些变量必须是 final 或 effectively final 的。这意味着这些变量一旦初始化后就不能被修改，这对于需要动态改变其值的游戏状态变量（如生命值和防御值）来说是个问题。

在游戏中，每个士兵都有自己的属性，如生命值和防御值，这些属性会随着游戏的进行（如遭受攻击）而变化。使用普通的 int 类型作为这些属性的存储会导致无法在事件监听器内部直接修改它们，因为在事件监听器中引用的局部变量必须是 final 或 effectively final 的。

### 2.解决方案

为了解决这个问题，我们选择了使用容器类型来存储这些值。具体来说，我们采用了如 AtomicInteger 这样的原子类，因为它们提供了线程安全的操作，同时允许在声明为 final 的情况下修改存储的数值。

### 3.结果

这种方法解决了在 GUI 事件监听器中动态修改状态变量的问题，同时保持了代码的整洁和效率。通过使用 AtomicInteger，我们不仅确保了线程安全，还克服了 Java 闭包的限制，使得游戏逻辑更加健壮和灵活。

这种策略最终使得游戏开发过程中的状态管理更加直接和高效，而且还提高了应用程序在处理并发操作时的可靠性。

即：

· **模块化**: 通过将功能分离到不同的块（数据处理、事件监听、GUI 更新），代码更加模块化，易于维护和扩展。

· **响应式 GUI**: 用户操作会即时反映在界面上，提高了用户体验。

· **线程安全**: 使用 AtomicInteger 确保即使在多线程环境中数值的增减也是安全的，避免了并发错误。

· **用户友好性**: 通过对话框和即时更新的 GUI 元素，提供了直观的反馈，增强了交互性。

### 4.实现细节

定义士兵类：每个士兵对象包含生命值和防御值，这些属性用 AtomicInteger 实现，以便可以在事件监听器中安全地进行更新。

#### 以Point de vie为例：

// 定义一个方法用于修改两个 AtomicInteger 类型的参数，根据布尔值 plus 来决定是增加还是减少

public void calculer(AtomicInteger nombrepoint, AtomicInteger value, boolean plus) {

if (plus) {

nombrepoint.getAndDecrement(); // 如果 plus 为 true，nombrepoint 减 1

value.getAndIncrement(); // value 加 1

} else {

nombrepoint.getAndIncrement(); // 如果 plus 为 false，nombrepoint 加 1

value.getAndDecrement(); // value 减 1

}

}

// 创建一个 AtomicInteger 类型的变量 pvalue，并初始化为 30

AtomicInteger pvalue = new AtomicInteger(30);

// 设置标签 PValue 的文本为 pvalue 的当前值，将 AtomicInteger 转换为字符串

PValue.setText(String.valueOf(pvalue.get()));

// 创建一个按钮 PPlusButton，用于增加点数

JButton PPlusButton = new JButton();

PPlusButton.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

calculer(nombrepoint, pvalue, true); // 调用 calculer 方法，传入 true 表示增加点数

NombrePointLable.setText(String.valueOf(nombrepoint.get())); // 更新 NombrePointLabel 的显示为 nombrepoint 的当前值

PValue.setText(String.valueOf(pvalue.get())); // 更新 PValue 标签的显示为 pvalue 的当前值

}

});

// 设置按钮的图标

PPlusButton.setIcon(new ImageIcon(PlayFrame.class.getResource("/imgs/plus.png")));

// 设置按钮的位置和大小

PPlusButton.setBounds(1206, 130, 30, 30);

// 将按钮添加到容器（通常是 JFrame 或 JPanel）中

getContentPane().add(PPlusButton);

// 创建一个按钮 PMoinsButton，用于减少点数

JButton PMoinsButton = new JButton("");

PMoinsButton.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

if (pvalue.get() >= 31) { // 检查 pvalue 是否大于或等于 31

calculer(nombrepoint, pvalue, false); // 调用 calculer 方法，传入 false 表示减少点数

NombrePointLable.setText(String.valueOf(nombrepoint.get())); // 更新 NombrePointLabel 的显示为 nombrepoint 的当前值

PValue.setText(String.valueOf(pvalue.get())); // 更新 PValue 标签的显示为 pvalue 的当前值

} else {

// 如果 pvalue 小于 31，显示一个信息对话框提示不能再减少

JOptionPane.showMessageDialog(null, "La valeur de la vie ne peut être inférieure à 30!", "Information", JOptionPane.INFORMATION\_MESSAGE);

}

}

});

// 设置按钮前景色

PMoinsButton.setForeground(new Color(240, 255, 240));

// 设置按钮的图标

PMoinsButton.setIcon(new ImageIcon(PlayFrame.class.getResource("/imgs/moins.png")));

// 设置按钮的位置和大小

PMoinsButton.setBounds(1085, 130, 30, 27);

// 将按钮添加到容器中

getContentPane().add(PMoinsButton);

· **函数** calculer: 这个函数根据传入的 plus 参数增加或减少 nombrepoint 和 value 的值。它使用了 AtomicInteger 的方法，以线程安全的方式修改数值。

· **按钮事件监听器**: PPlusButton 和 PMoinsButton 按钮通过事件监听器调用 calculer 函数。这允许用户界面与数据逻辑分离，使得用户操作直接影响数据的修改。

· **数据和显示的同步更新**: 按钮的动作不仅改变数据，还立即更新 GUI 上的显示，保证显示的数据是最新的。

## 二、视图跳转

##### 1.问题描述

在最初的游戏开发阶段，我们选择使用多个独立的 JFrame 来处理不同游戏阶段的界面跳转。这种方法虽然在逻辑上简单明了，但在实际操作中带来了一系列问题：

资源消耗：每个独立的 JFrame 都需要单独的资源加载，这在资源有限的环境下会影响游戏性能。

用户体验：多窗口切换对用户来说可能会造成混淆，尤其是在游戏关键时刻需要快速反应时，窗口之间的切换可能会打断玩家的沉浸感。

维护难度：多个 JFrame 的管理和维护工作量大，每个窗口的状态管理也相对复杂。

##### 2.解决方案

为了解决这些问题，我们希望能够放弃使用多个独立的 JFrame，转而采用单一窗口配合 CardLayout 管理多个面板的方法。CardLayout 允许在同一个窗口内部通过卡片式布局来切换不同的界面，每张卡片代表游戏的一个不同阶段，更适合我们的项目。但是因为时间有限，我们还未来得及实施。

布局转换：

首先，我们可以将原有的多个 JFrame 转换为一个主 JFrame，里面使用 CardLayout。

将每个原本独立的界面转换为一个 JPanel，并被添加到主 JFrame 的 CardLayout 中。

资源优化：

所有需要预加载的资源现在都在游戏启动时一次性加载，降低了运行时的资源消耗。

通过共享资源，如字体、图像和声音文件，减少了内存的占用。

界面切换优化：

如果按照预想继续改善，切换界面只需调用 CardLayout 的 show() 方法，即可实现无缝切换。可以增加界面切换的动画效果，并提升用户体验。

##### 3.结果

如果能够解决这个问题，将会极大地提升应用的性能和用户体验：

性能提升：由于减少了重复的资源加载，应用的响应时间更快，运行更加流畅。

用户体验改善：用户可以在一个连贯的环境中游戏，无需跳转到不同的窗口，这可以提升游戏的沉浸感和操作的便捷性。

维护简化：单一窗口和统一的资源管理使得后续的维护和更新更加简单高效。

总体来说，通过将多个独立的 JFrame 替换为一个使用 CardLayout 的单一 JFrame，我们可以解决多窗口带来的性能和用户体验问题。这种设计方法不仅可以优化资源的使用和管理，还可以提升整个应用的专业性和用户的满意度。虽然还未完全实现，但是这一经验也为我们后续的项目设计提供了宝贵的参考，在遇到类似问题时，我们可以直接考虑采用单一窗口配合 CardLayout 管理多个面板的方法进行设计。

## 三、界面视觉风格统一

在使用 WindowBuilder 进行游戏界面设计时，我们遇到了一个常见的挑战：界面控件的默认样式与游戏的整体风格不一致。这是因为 WindowBuilder 中提供的默认控件（如按钮）通常具有标准的视觉样式，这些样式往往较为简单且功能性强，但可能缺乏为特定游戏环境设计的视觉吸引力和主题一致性。

### 1.问题描述

在开发一个适合所有年龄的角色扮演游戏时，我们希望游戏的用户界面能够反映出这一环境，包括使用圆滑的边框、卡通的图像、易理解的图标等元素。然而，使用 WindowBuilder 创建的默认按钮通常是现代风格的平面或轻微的三维按钮，这与游戏的沉浸式体验相去甚远。

### 2.解决方案

为了解决这个问题，我们团队进行了一系列的思考和讨论。最终，我们决定采用将控件的视觉元素自定义的方法，特别是通过更改按钮和标签的图标来匹配游戏的主题风格。以下是我们采取的具体步骤：

##### 资源收集和制作：

我们首先从各种资源库和手动绘画的方式收集和创作与游戏主题相符的图像素材，如中世纪形象的三种士兵、各种按钮图案和符文。

##### 设计自定义图标：

接着，我们利用图形设计软件（如 Photoshop 或 Illustrator）将这些图像素材调整到适合的尺寸和格式，并设计成符合游戏风格的按钮图标。

##### 实施到 WindowBuilder：

在 WindowBuilder 中，我们将默认的按钮图标替换为这些自定义的图标。通过设置按钮的 setIcon() 方法，将图标应用到界面元素上。

例如：button.setIcon(new ImageIcon("path/to/background\_icon.png"));

##### 细节调整和测试：

完成图标替换后，我们对游戏界面进行了详细的测试，以确保所有控件在不同状态下（如点击）的视觉效果和功能表现仍然符合预期。

##### 反馈循环：

在初步测试后，我们收集了目标用户的反馈，并根据反馈进一步调整控件的视觉细节，以最大程度地提升用户体验。

### 3.结果与影响

通过将标准控件转换为符合游戏主题的自定义图标，我们不仅增强了游戏的视觉吸引力，还提升了玩家的沉浸感。这种方法证明，即使是使用标准的 GUI 开发工具，通过创造性的解决方案也能有效地整合游戏设计元素，从而达到既美观又功能全面的界面设计。这一过程还加深了团队对于资源的管理、图形设计及用户接口设计的理解和技能。

## 四、横向滚动条问题

在游戏开发过程中，尤其是涉及到用户界面（UI）的设计时，经常会遇到需要根据实际使用体验调整设计初衷的情况。在我们的开发过程中，一个典型的例子是处理游戏中的士兵角色选择界面，如何在保持界面美观的同时，也让玩家能够方便地操作和浏览所有选项。

### 1.问题描述

在设计游戏 "Paramétrage des troupes" 的士兵选择界面时，最初的想法是使用一个横向滚动条来展示所有可选的士兵（如图1）。这样设计的目的是为了节省空间，同时允许玩家通过滚动条浏览所有士兵选项。

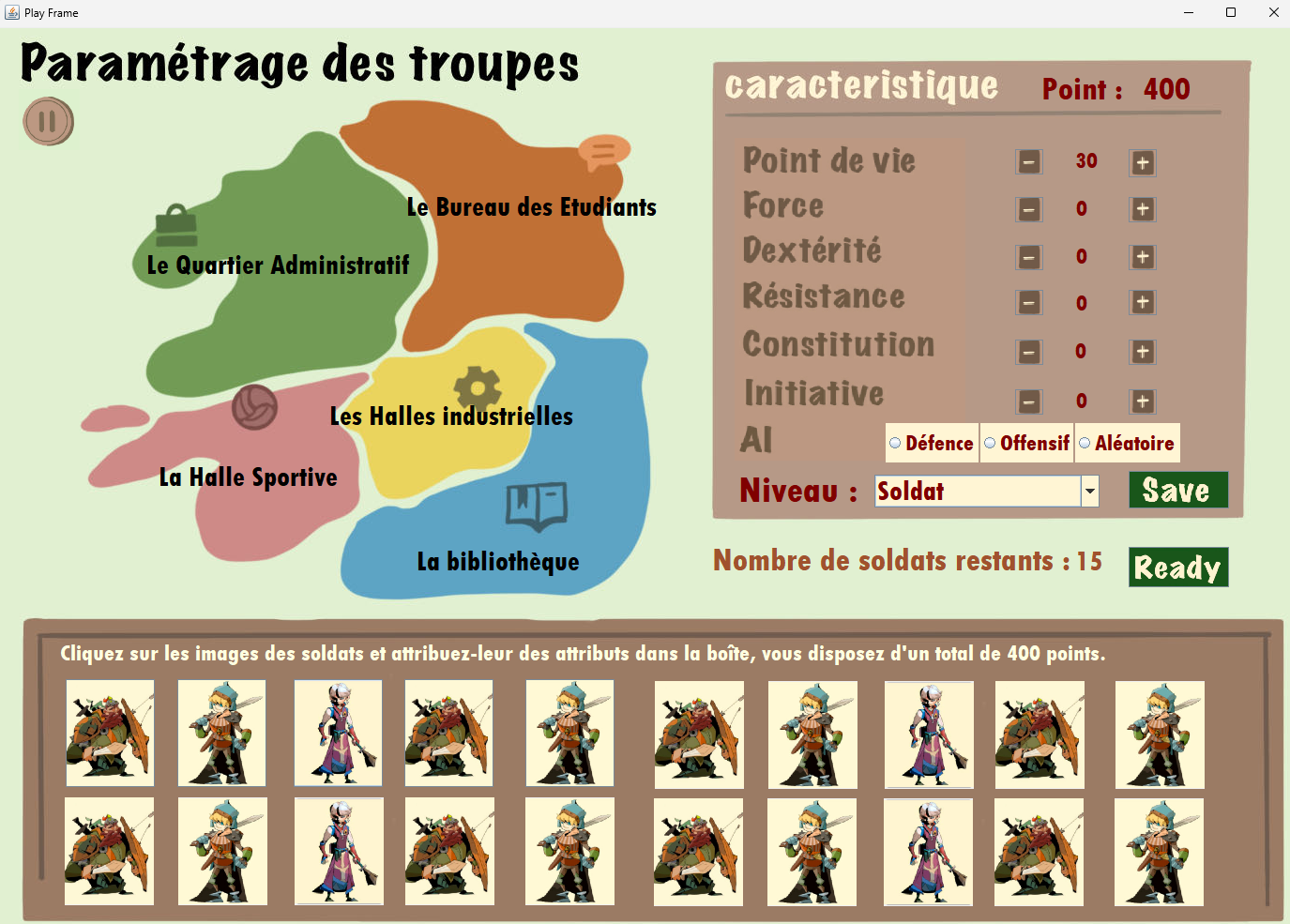


（图1）

然而，在实际开发和用户测试过程中，我们发现这种设计虽然节省了空间，但却降低了界面的直观性和易用性。士兵的图标和信息在横向滚动条中显得拥挤，玩家很难快速识别和选择他们想要的士兵，尤其是当士兵种类较多时。

### 2.解决方案

为了解决这个问题，我们决定放弃使用横向滚动条的方案，转而采用将所有士兵直接在界面下方展示的方法（如图2)。这一改变使得每个士兵的图标都能清晰地显示在界面上，玩家可以一眼看到所有可选的士兵，从而做出更快的决策。



（图2）

### 3.实施细节

我们对交互设计进行了优化，对于屏幕较小的设备，通过优化图标的排列和间距，确保操作的便捷性，同时增加了士兵选择的响应速度和反馈效果，较为简约明了的布局了各种士兵和对应属性。

### 4.结果

这一改变极大地提升了用户界面的友好度和游戏的整体可玩性。玩家可以快速、直观地选择士兵，而不需要在滚动条和拥挤的图标中挣扎。此外，从用户反馈来看，玩家对这种直接展示所有选项的方式更加满意，认为这使得游戏更加吸引人。

总的来说，在这个过程中我们明白游戏界面设计中根据实际用户体验进行迭代的重要性。通过对设计方案的持续评估和调整，我们希望提供更符合用户需求的解决方案，从而提升整个游戏的质量和玩家的满意度。

## 四、背景图片自适应窗口大小问题

### 问题描述

在项目开发过程中，我们通常会希望背景图片能够随着窗口大小的变化而自动调整尺寸。这一需求在我们的实际开发过程中遇到了一些困难，因为默认情况下，Swing组件并不具备自动调整背景图片大小的功能。

具体来说，我们想要实现以下功能：

1. 背景图片能够随着JFrame窗口的大小变化而自动调整，以始终覆盖整个窗口。

2. 在窗口大小改变时，背景图片能够即时更新，而不会出现显示错误或延迟。

### 解决方案

由于时间较为短暂、技术不够成熟等问题，我们暂时还未能解决这个问题。但是根据我们的讨论和研究，基本有一些思考与计划。

为了解决上述问题，我们希望能够采用以下步骤：

步骤一：重写JLabel的paint方法

首先，我们可以通过继承JLabel并重写其paint方法来实现背景图片的动态调整。重写的paint方法将根据JLabel的当前大小绘制背景图片。

例如：

ImageIcon ii = new ImageIcon("bg2.jpg"); // 加载背景图片

JLabel bg = new JLabel(ii) {

@Override

public void paint(Graphics g) { // 匿名类重写paint方法

Image img = ii.getImage();

g.drawImage(img, 0, 0, this.getWidth(), this.getHeight(), null);

}

};

我们希望通过创建一个匿名类继承自JLabel，并重写了paint方法的方式实现我们的愿景。

步骤二：添加ComponentListener监听器

为了确保背景图片在窗口大小改变时能够自动调整，我们应该还会需要为JFrame添加一个ComponentListener监听器，为了监视窗口大小的变化，并在变化时重新设置JLabel的大小和重新绘制背景图片。其具体实现暂未明晰，但是我们希望能够通过如下类似的方式进行实现：

this.addComponentListener(new ComponentAdapter() {

@Override

public void componentResized(ComponentEvent e) { // 监听窗口大小变化

bg.setSize(this.getSize()); // 重新设置JLabel大小

bg.repaint(); // 重新绘制背景

}

});

在这个监听器中，我们希望`componentResized`方法能够在窗口大小改变时被调用。为了实现这个目的，我们可以在该方法中重新设置JLabel的大小为窗口的当前大小，并调用`bg.repaint()`方法强制JLabel重新绘制。

### 结果

- \*\*图像质量\*\*：由于在调整图片大小时可能会影响图像的质量，导致玩家游戏体验收到影响，因此这个方法还待优化和改进。

- \*\*性能问题\*\*：频繁调整窗口大小可能会导致频繁重绘，影响性能，可能出现游戏过程中出现卡顿等情况，因此暂时将背景图片随着窗口大小的变化而自动调整的尝试搁置并期待今后对这像功能的实现。