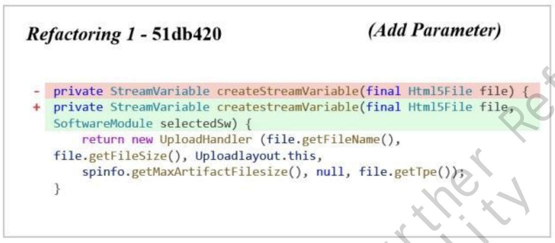
Potential Causes of Refactoring Evolution Volatility

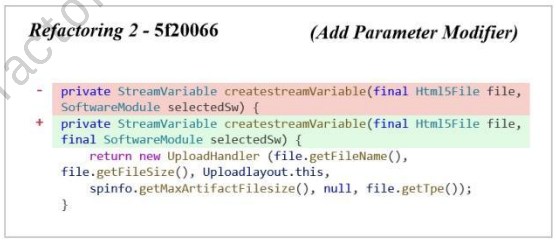
Causes 1: Potential Issue Addressing

Definition: This involves addressing potential issues or technical debt that were not fully resolved during the initial refactoring. Developers may identify latent issues that could lead to bugs or code structure erosion if left unaddressed. Therefore, subsequent refactoring is necessary to mitigate these risks and ensure code stability and quality.

Example: Figure below shows a representative example to illustrate this reason. In the first commit 51db420, the parameter SoftwareModule selectedSw is added to the method createStreamVariable(). However, omitting the final modifier could lead developers to mistakenly believe that the parameter is modifiable within the method, potentially causing inadvertent changes during future maintenance and introducing hard-to-detect errors. The subsequent commit 5f20066 addresses this by adding the final modifier to SoftwareModule selectedSw, rectifying the oversight. This safeguard ensures the parameter cannot be modified within the method, preventing potential bugs and enhancing code security and readability.


Refactoring 1* - 51db420** ***(Add Parameter)

```
- private StreamVariable createStreamVariable(final Html5File file) {
+ private StreamVariable createstreamVariable(final Html5File file,
  SoftwareModule selectedSw) {
      return new UploadHandler (file.getFileName(),
  file.getFileSize(), Uploadlayout.this,
      spinfo.getMaxArtifactFilesize(), null, file.getTpe());
  }
```


Refactoring 2* - 5f20066** ***(Add Parameter Modifier)

```
- private StreamVariable createstreamVariable(final Html5File file,
  SoftwareModule selectedSw) {
+ private StreamVariable createstreamVariable(final Html5File file,
  final SoftwareModule selectedSw) {
      return new UploadHandler (file.getFileName(),
  file.getFileSize(), Uploadlayout.this,
      spinfo.getMaxArtifactFilesize(), null, file.getTpe());
  }
```

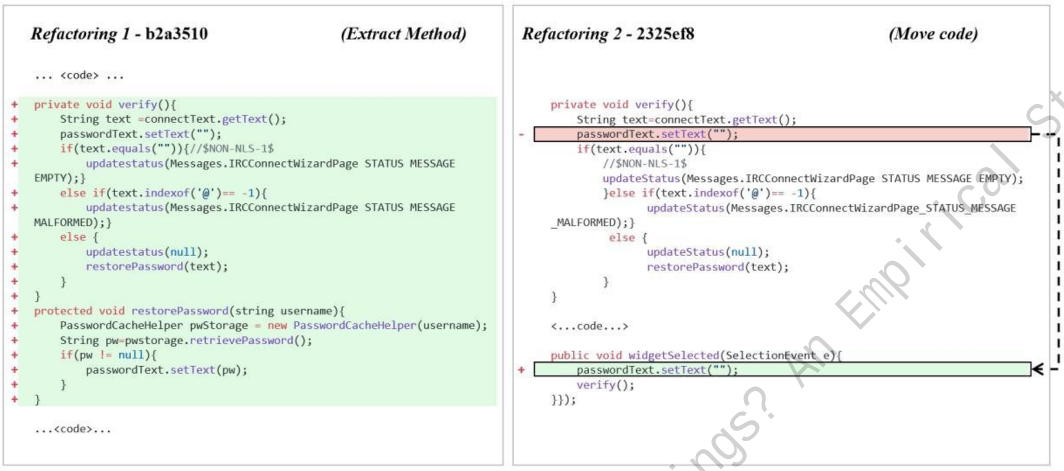
A real example is explained：Potential Issue Addressing

Causes 2: Bug Fixing

Definition: Refactorings can inadvertently introduce bugs into the codebase. These bugs may not be immediately apparent and can surface after the initial refactoring. When developers detect these bugs, they need to perform additional refactoring to fix them, ensuring the functionality and reliability of the software.

Example: The Figure below illustrates a real example of the bug-fixing category from the Ecf project. In the first refactoring commit b2a3510, the developer extracted two methods, verify() and restorePassword(). The verify() method checks input validity and calls restorePassword() to retrieve and set the password in the passwordText field. However, the line passwordText.setText("") in verify() cleared the password text, preventing it from being displayed correctly. In the second refactoring commit 2325ef8, the developer extracted the problematic line from verify() and placed it in widgetSelected(). This change ensured the password field was cleared appropriately, not during verification. This is a targeted password clearing operation. The program only clears the password by executing passwordText.setText("") when the widgeSelected() method is called, not every time verify() is executed. Moving the statement passwordText.setText("") out of verify() prevents the password from being emptied incorrectly.

Finally, this bug was fixed by adjusting the location of the cleared password to ensure that the password text box is only cleared at the appropriate time, preventing improper overwriting. Overall, the first refactoring enhanced the application by adding password storage functionality but inadvertently introduced an error related to the unnecessary clearing of the password field. The second refactoring fixed this by adjusting the timing and method of clearing the password field, thereby improving the handling of password caching.



***Refactoring 1* - b2a3510** *(Extract Method)*

```
... <code> ...

+ private void verify(){
+     String text =connectText.getText();
+     passwordText.setText("");
+     if(text.equals("")){//$NON-NLS-1$
+         updatestatus(Messages.IRCConnectWizardPage STATUS MESSAGE
EMPTY);}
+     else if(text.indexof('@')== -1){
+         updatestatus(Messages.IRCConnectWizardPage STATUS MESSAGE
MALFORMED);}
+     else {
+         updatestatus(null);
+         restorePassword(text);
+     }
+ }
+ protected void restorePassword(string username){
+     PasswordCacheHelper pwStorage = new PasswordCacheHelper(username);
+     String pw=pwstorage.retrievePassword();
+     if(pw != null){
+         passwordText.setText(pw);
+     }
+ }

...<code>...
```

***Refactoring 2* - 2325ef8** *(Move code)*

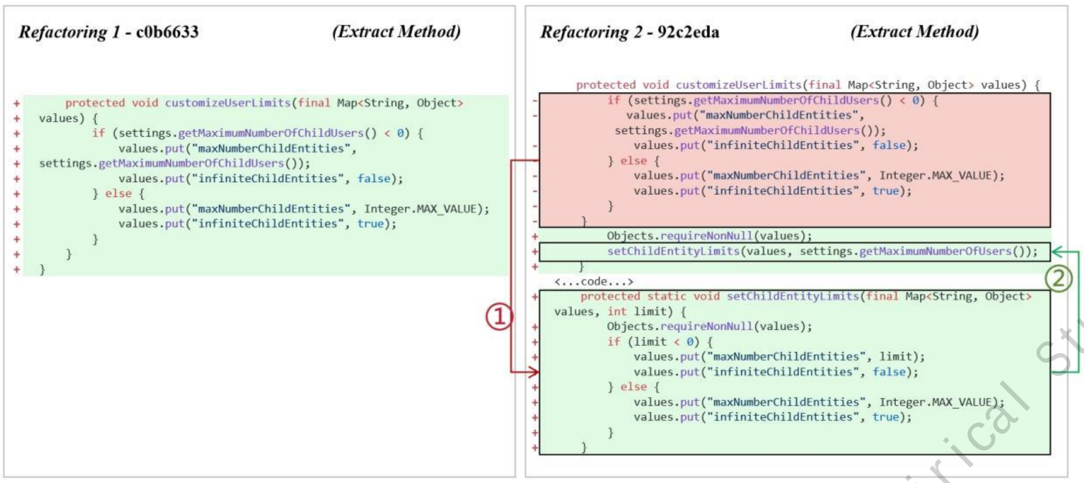
```
  private void verify(){
      String text=connectText.getText();
-     passwordText.setText("");
      if(text.equals("")){
          //$NON-NLS-1$
          updateStatus(Messages.IRCConnectWizardPage STATUS MESSAGE EMPTY);
          }else if(text.indexof('@')== -1){
                updateStatus(Messages.IRCConnectWizardPage_STATUS_MESSAGE
_MALFORMED);}
            else {
                updateStatus(null);
                restorePassword(text);
          }
  }

  <...code...>

  public void widgetSelected(SelectionEvent e){
+     passwordText.setText("");
      verify();
  }});
```

A real example is explained the cause：Bug Fixing

Causes 3: Structure Refining

Definition: This indicates that the refactoring action did not achieve optimal results, failing to significantly improve the code structure quality and sometimes making the code less clear. Consequently, further refactoring is necessary to optimize the code and enhance its quality. Unlike potential issue addressing, the refactoring to be optimized typically does not involve issues that could lead to defects or threaten software reliability.

Example: The Figure below shows another example from the Kapua project involving commit c0b6633, where developers extracted the logic for configuring resource limits in the user service into a separate customizeUserLimits() method (①). By extracting this method, the developers have abstracted the details from the previous method into an independent functional module, reducing code duplication and improving readability. Additionally, by utilizing the configuration from settings, the maximum number of child users can be flexibly adjusted based on the application environment, enhancing the configurability and scalability of code. Subsequently, in commit 92c2eda, the developers further refactored the customizeUserLimits() method by extracting a new static method, setChildEntityLimits(), which consolidates the logic related to setting child entity limits(②). By extracting the setChildEntityLimits() method, the redundant logic for setting child entity limits was eliminated, and the structure of the code is further optimized. This refactoring, through method extraction and parameterization, improved code reusability, scalability, and maintainability while reducing redundant code. In any context where child entity limits need to be set, the logic can now be uniformly managed by invoking the setChildEntityLimits() method.



```
+     protected void customizeUserLimits(final Map<String, Object>
+ values) {
+         if (settings.getMaximumNumberOfChildUsers() < 0) {
+             values.put("maxNumberChildEntities",
+ settings.getMaximumNumberOfChildUsers());
+             values.put("infiniteChildEntities", false);
+         } else {
+             values.put("maxNumberChildEntities", Integer.MAX_VALUE);
+             values.put("infiniteChildEntities", true);
+         }
+     }
+ }
```

```
      protected void customizeUserLimits(final Map<String, Object> values) {
-         if (settings.getMaximumNumberOfChildUsers() < 0) {
-            values.put("maxNumberChildEntities",
          settings.getMaximumNumberOfChildUsers());
-             values.put("infiniteChildEntities", false);
-         } else {
-             values.put("maxNumberChildEntities", Integer.MAX_VALUE);
-             values.put("infiniteChildEntities", true);
-         }
-     }
+         Objects.requireNonNull(values);
+         setChildEntityLimits(values, settings.getMaximumNumberOfUsers());
+     }
  <...code...>
+     protected static void setChildEntityLimits(final Map<String, Object>
  values, int limit) {
+         Objects.requireNonNull(values);
+         if (limit < 0) {
+             values.put("maxNumberChildEntities", limit);
+             values.put("infiniteChildEntities", false);
+         } else {
+             values.put("maxNumberChildEntities", Integer.MAX_VALUE);
+             values.put("infiniteChildEntities", true);
+         }
+     }
```

A real example is explained the cause：Structure Refining

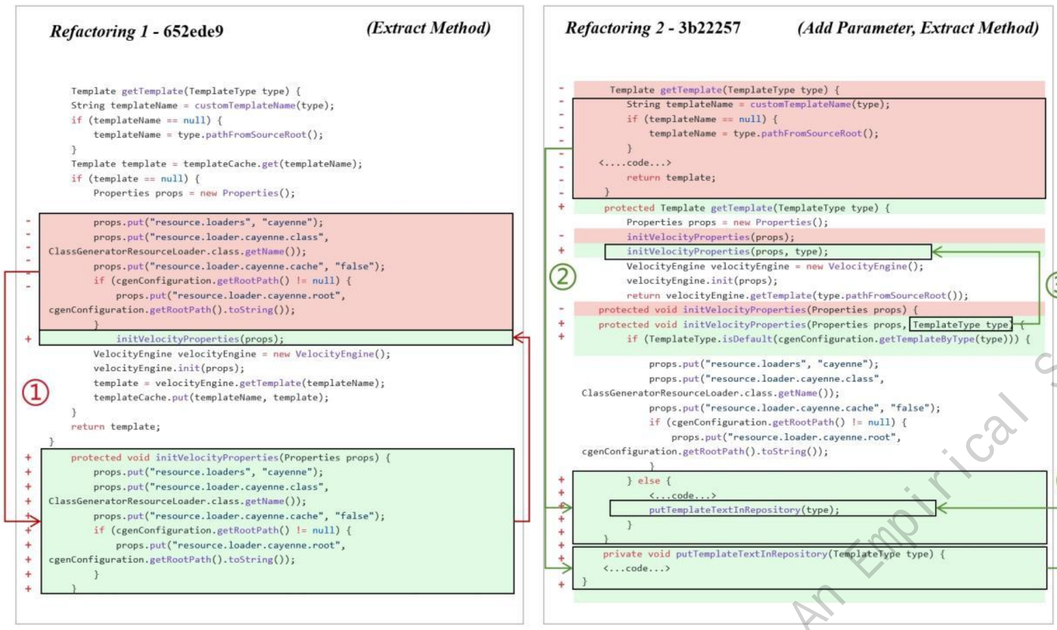
Causes 4: Feature Updating

Definition: Feature addition indicates that the developer is adding or enhancing a feature, and this common reason for refactoring volatility stems from the need to add a new feature to a previous refactoring while improving the quality of that part of the code.

Example: The Figure below reports an re-refactoring instance from the project Cayenne illustrating this reason, where the first refactor commit 652ede9 conducted Extract Method for the file ClassGenerationAction.java to generate the method initVelocityProperties(). This refactoring centralized the logic that was originally used to configure the initialization of the VelocityEngine within the getTemplate() method has been extracted into a new method, initVelocityProperties() (①). By extracting the configuration logic to the initVelocityProperties() method, the logic of the getTemplate() method has been simplified and made more homogeneous - it is primarily responsible for handling template fetching and caching. This reduces the complexity of the methods and makes the code more readable and easier to understand and maintain.

Subsequently, the new workflow was introduced in commit 3b22257, prompting developers to refactor the code again to accommodate these updates. This refactoring optimizes the template acquisition and processing logic in the getTemplate() method and introduces new workflows. First, by Extract Method, the template processing logic in the getTemplate() method was extracted to the initVelocityProperties() and putTemplateTextInRepository() method, centralized management and dynamic adjustment of the configuration is achieved. Second, by adding a new parameter TemplateType type to the initVelocityProperties() method, initVelocityProperties() can automatically adapt and apply the corresponding configuration parameters according to the template type, implementing a differentiated loading strategy, which greatly enhances the flexibility and adaptability of template processing. This mechanism greatly enhances the flexibility and adaptability of template processing.

Compared to the earlier refactoring, this not only streamlines the complexity of the getTemplate() method, but also improves code maintainability and introduces new features to the template handling module.



Refactoring 1 - 652ede9 *(Extract Method)*

```
    Template getTemplate(TemplateType type) {
    String templateName = customTemplateName(type);
    if (templateName == null) {
        templateName = type.pathFromSourceRoot();
    }
    Template template = templateCache.get(templateName);
    if (template == null) {
        Properties props = new Properties();
-       props.put("resource.loaders", "cayenne");
-       props.put("resource.loader.cayenne.class",
- ClassGeneratorResourceLoader.class.getName());
-       props.put("resource.loader.cayenne.cache", "false");
-       if (cgenConfiguration.getRootPath() != null) {
-           props.put("resource.loader.cayenne.root",
  cgenConfiguration.getRootPath().toString());
        }
+           initVelocityProperties(props);
        VelocityEngine velocityEngine = new VelocityEngine();
        velocityEngine.init(props);
        template = velocityEngine.getTemplate(templateName);
        templateCache.put(templateName, template);
    }
    return template;
  }
+   protected void initVelocityProperties(Properties props) {
+       props.put("resource.loaders", "cayenne");
+       props.put("resource.loader.cayenne.class",
+ ClassGeneratorResourceLoader.class.getName());
+       props.put("resource.loader.cayenne.cache", "false");
+       if (cgenConfiguration.getRootPath() != null) {
+           props.put("resource.loader.cayenne.root",
+ cgenConfiguration.getRootPath().toString());
+       }
+   }
```

Refactoring 2 - 3b22257 *(Add Parameter, Extract Method)*

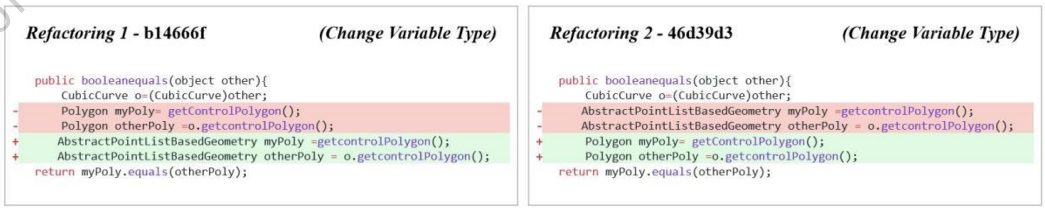
```
-    Template getTemplate(TemplateType type) {
-       String templateName = customTemplateName(type);
-       if (templateName == null) {
-           templateName = type.pathFromSourceRoot();
-       }
-   <....code...>
-       return template;
-    }
+   protected Template getTemplate(TemplateType type) {
        Properties props = new Properties();
-       initVelocityProperties(props);
+       initVelocityProperties(props, type);
        VelocityEngine velocityEngine = new VelocityEngine();
        velocityEngine.init(props);
        return velocityEngine.getTemplate(type.pathFromSourceRoot());
-   protected void initVelocityProperties(Properties props) {
+   protected void initVelocityProperties(Properties props, TemplateType type) {
+       if (TemplateType.isDefault(cgenConfiguration.getTemplateByType(type))) {

            props.put("resource.loaders", "cayenne");
            props.put("resource.loader.cayenne.class",
 ClassGeneratorResourceLoader.class.getName());
            props.put("resource.loader.cayenne.cache", "false");
            if (cgenConfiguration.getRootPath() != null) {
                props.put("resource.loader.cayenne.root",
 cgenConfiguration.getRootPath().toString());
            }
+       } else {
+           <...code...>
+           putTemplateTextInRepository(type);
+       }
+   }
+   private void putTemplateTextInRepository(TemplateType type) {
+   <...code...>
+ }
```

A real example is explained the cause： Feature Updating

Causes 5: Feature Reverting

Definition: Feature reverting occurs when the previous refactoring introduces some new features or functionality, and the developer removes them in the subsequent refactoring to accommodate changing requirements.

Example: The Figure below shows a real example of Feature Reverting. In this example, there is a commit b14666f from the Gef project, where a refactoring was performed to enhance code flexibility and extensibility. The developer introduced AbstractPointListBasedGeometry to reduce duplicated code. To accommodate this change, the developer changed the type of the variables myPoly and otherPoly from Polygon to AbstractPointListBasedGeometry. The main purpose of this refactoring is to replace the concrete class Polygon with the more generic abstract class AbstractPointListBasedGeometry. This allowed the getControlPolygon() method to return a more generic type, thereby improving flexibility. However, within two months, another commit 46d39d3 reverted this change, refactoring the Geometry API and changing the variable types back to Polygon. This reversal might have occurred because Polygon better suited the specific business logic or simplified the code by avoiding type conversions and reducing the overhead associated with abstract classes.



Refactoring 1 - b14666f *(Change Variable Type)*

```
  public booleanequals(object other){
      CubicCurve o=(CubicCurve)other;
-     Polygon myPoly= getControlPolygon();
-     Polygon otherPoly =o.getcontrolPolygon();
+     AbstractPointListBasedGeometry myPoly =getcontrolPolygon();
+     AbstractPointListBasedGeometry otherPoly = o.getcontrolPolygon();
  return myPoly.equals(otherPoly);
```

Refactoring 2 - 46d39d3 *(Change Variable Type)*

```
  public booleanequals(object other){
      CubicCurve o=(CubicCurve)other;
-     AbstractPointListBasedGeometry myPoly =getcontrolPolygon();
-     AbstractPointListBasedGeometry otherPoly = o.getcontrolPolygon();
+     Polygon myPoly= getControlPolygon();
+     Polygon otherPoly =o.getcontrolPolygon();
  return myPoly.equals(otherPoly);
```

A real example is explained the cause： Feature Reverting