TechMatrix

テスト自動化基盤をどう作るか

実践！ソースコードのバグ検出
～失敗例から見るテスト自動化のポイントとは～

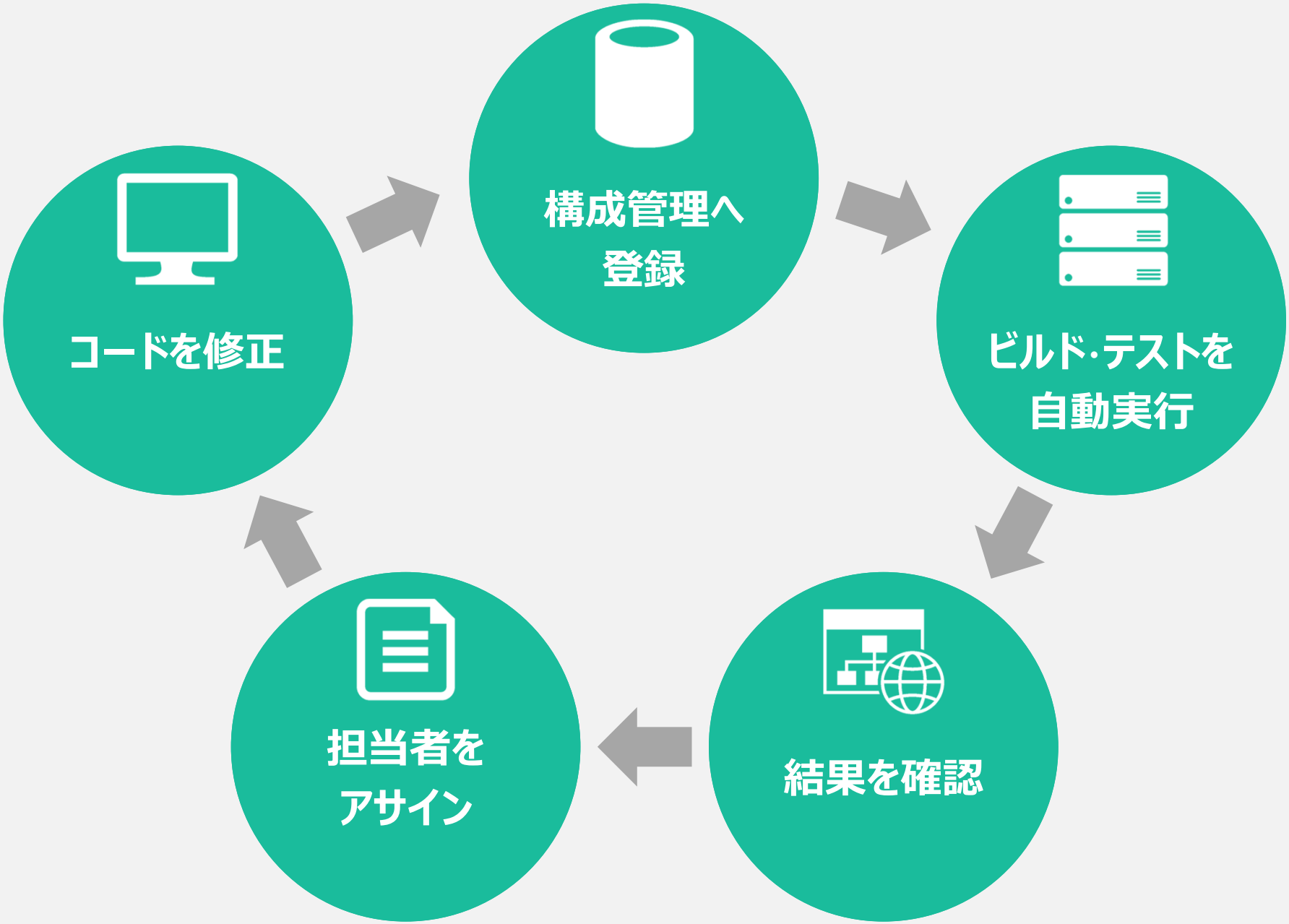
継続的インテグレーション

継続的インテグレーション、CI（英: continuous integration）とは、
主にプログラマーのアプリケーション作成時の品質改善や納期の短縮のための習慣のことである。

エクストリーム・プログラミング (XP) のプラクティスの一つで、狭義にはビルドやテスト、
インスペクションなどを継続的に実行していくことを意味する。

- Wikipedia「継続的インテグレーション」より引用

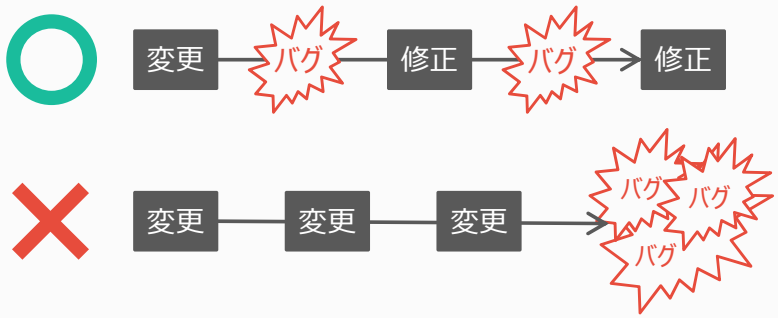
継続的インテグレーションを利用した開発の流れ



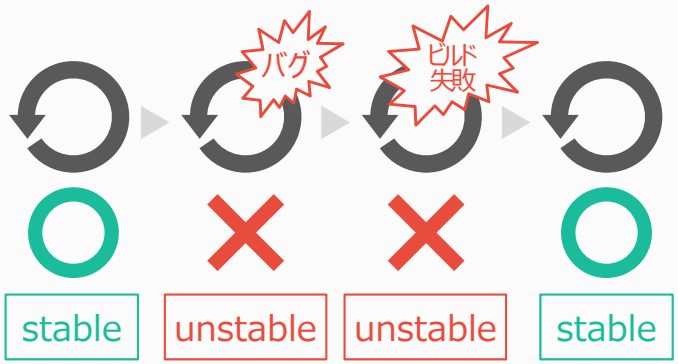
継続的インテグレーションのメリット

メリット

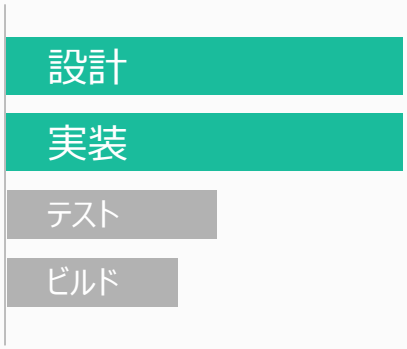
短いサイクルでバグを取り除く



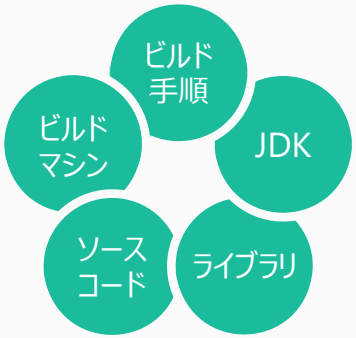
安定版を確保する



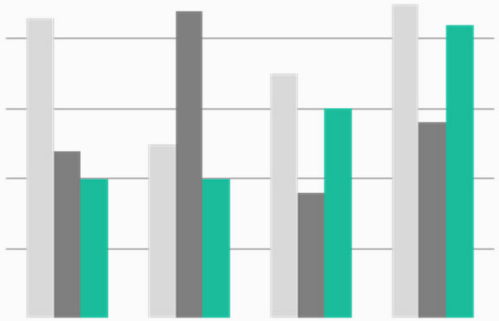
知的な作業に集中する



同じ環境・手順・設定でテストする



開発データを蓄積する



継続的インテグレーションで行われる「ビルド」や「テスト」

継続的インテグレーションで用いられるツールの例

継続的インテグレーションの要素		用いられるツール
継続的インテグレーションツール		**Jenkins / TravisCI**
構成管理		**Git / Subversion / AccuRev**
ビルドの自動化		**Ant / Maven / Gradle**
テストの自動化	静的解析	**Jtest / SpotBugs / CheckStyle**
	単体テスト	**Jtest / Junit / djUnit**
	結合テスト	**Ranorex / Selenium**
デプロイ		**Ant / Maven / Gradle**

テスト自動化で静的解析を行いバグを見つける

- 今回は**静的解析**をテスト自動化にて実施した際の失敗例と改善のポイントをご説明します。

失敗例を含んだ開発プロジェクト

ある開発プロジェクトのテスト自動化を取り上げ、テスト自動化の失敗例から改善点をご紹介します。

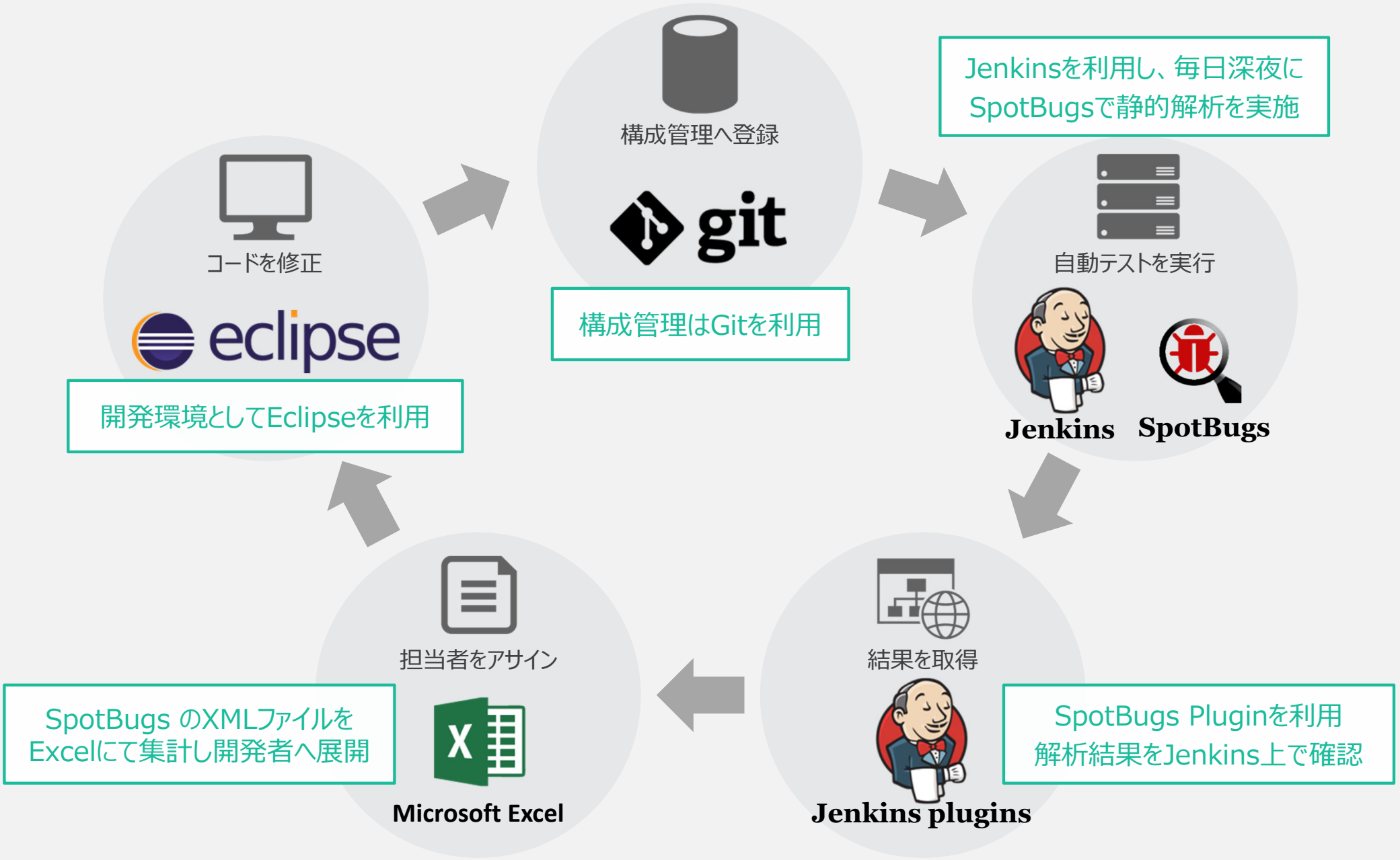
プロジェクトの概要

- 開発ソフトウェア：某Webシステム
- コード規模：約20万行
- 利用言語：Java

テスト自動化の概要

- オープンソースツールを利用したテスト自動化の環境を構築済みです。
- 毎日深夜に自動テストが実行されます。
- 自動テストではビルドの確認およびバグの検出を行います。

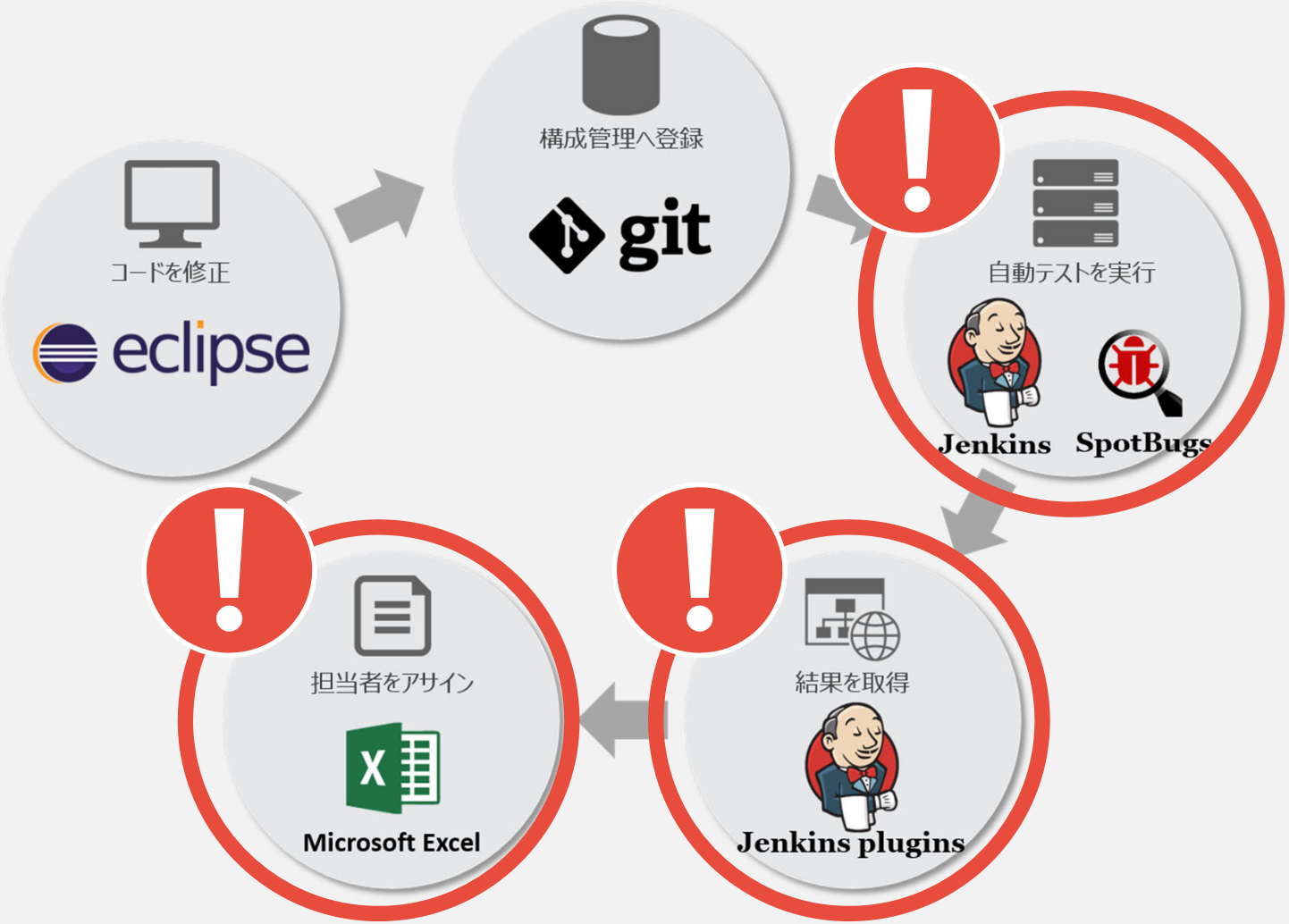
開発プロジェクトの開発の流れと利用ツール



問題の箇所

自動化の問題

- このプロジェクトのテスト自動化には複数の問題が存在しており、効果が出ていません。
- 具体的な3つの失敗例を取り上げ、効果的にテスト自動化を運用する方法をご紹介します。



失敗例1.

期待するバグが検出されない。

失敗例1. 期待するバグが検出されない。

自動テストを実行

- 毎日深夜にSpotBugsにて静的解析を実行している。
- 致命的な問題につながる可能性のある実装箇所を検出したい。
- 思ったようにバグが検出されず、機能テスト以降の工程で不具合が検出されてしまう。

ツールの解析精度に問題がある。

静的解析のおさらい

静的解析についておさらいします。

静的解析とは

- **ソースコードを入力とし、コードの実装内容や構造からさまざまなデータを分析します。**
- 広義な静的解析としては以下があります。

不具合につながる可能性の高い箇所の検出	ソフトウェア品質（メトリクス）の計測	ソフトウェア実行時のエラー検出

- また、ソースコードの解析方法には以下があります。

パターンマッチ解析（字句と構文から分析）	データ・構造フロー解析（バイトコードからデータや処理の流れを分析）

今回の開発プロジェクトでは

- **致命的な問題につながる可能性の高い箇所を検出**することをテストの自動化の目的としており、**フロー解析**が可能なSpotBugsを採用しています。

主な静的解析ツールの特徴をご紹介した上で、問題点の解説を行います。

Java向けの静的解析ツールの例

Java開発にて利用される静的解析ツールを以下に記載します。

Java向けの静的解析ツールの例

ツール名	特徴	
SpotBugs	潜在的なバグの検出に特化。バイトコードを解析（フロー解析）。	無償
CheckStyle	コーディング規約チェックに関するパターンが充実。	無償
PMD	潜在的なバグや規約チェック、メトリクスを幅広く検出。	無償
Error Prone	一部のコーディング記述ミスを検出する。Googleが開発。	無償
Jtest	コーディング規約チェック、潜在的なバグの検出、メトリクス計測に対応。 バイトコードを解析（フロー解析）。	有償

SpotBugsでバグを検出する…ただし

- SpotBugsはオープンソースの解析ツールの中で潜在的なバグの検出に特化しており、テスト自動化にてバグを検出する目的に合ったツールです。
- ただし、実プロジェクトでの利用を考えた場合は、SpotBugsのバグの検出能力に　　不安があります。
- フロー解析の性能について、**SpotBugs** と **Jtest**の具体的な例を交えてご説明します。

SpotBugs と Jtest® の検出能力の比較

NullPointerExceptionを引き起こすコード

以下のJavaコードにはNullPointerExceptionを引き起こすコードが含まれています。

ToupperCall.java

```java
package jp.co.techmatrix;

public class ToupperCall {
  public static void main( String[] args ) {
    Toupper name = new Toupper( 10 );
    System.out.println( name.toUpper() );
  }
```

Toupper.java

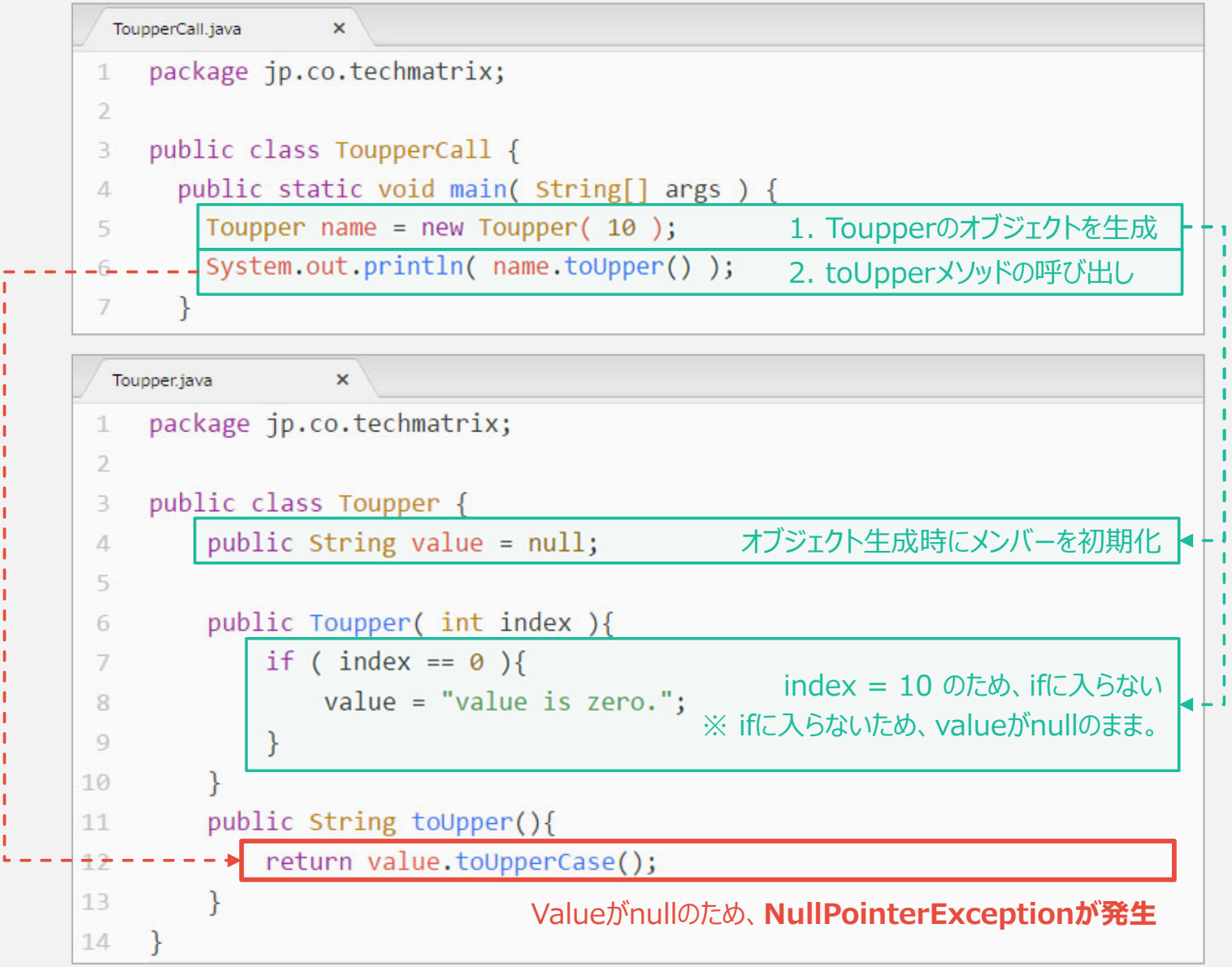
```java
package jp.co.techmatrix;

public class Toupper {
    public String value = null;

    public Toupper( int index ){
        if ( index == 0 ){
            value = "value is zero.";
        }
    }
    public String toUpper(){
        return value.toUpperCase();
    }
}
```

NullPointerExceptionを引き起こすコード 問題のある処理

NullPointerExceptionを引き起こす処理の流れは以下のとおりです。



NullPointerExceptionを引き起こすコードをSpotBugsで解析

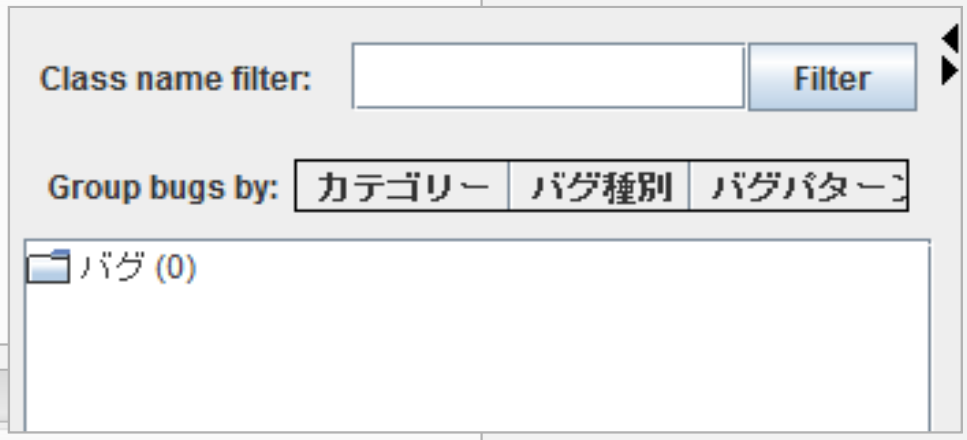
SpotBugs

検出できない

ToupperCall.java

```java
package jp.co.techmatrix;

public class ToupperCall {
  public static void main( String[] args ) {
    Toupper name = new Toupper( 10 );
    System.out.println( name.toUpper() );
  }
```



Toupper.java

```java
package jp.co.techmatrix;

public class Toupper {
    public String value = null;

    public Toupper( int index ){
        if ( index == 0 ){
            value = "value is zero.";
        }
    }
    public String toUpper(){
        return value.toUpperCase();
    }
}
```

SpotBugsはバイトコードを解析しますが、
解析範囲が1ファイル内(1クラス内)に限定されます。

NullPointerExceptionを引き起こすコードをJtestで解析

Jtest®

○ 検出する

ToupperCall.java

```
1  package jp.co.techmatrix;
2
3  public class ToupperCall {
4    public static void main( String[] arg
5      Toupper name = new Toupper( 10 );
6      System.out.println( name.toUpper()
7    }
```

Toupper.java

```
1  package jp.co.techmatrix;
2
3  public class Toupper {
4      public String value = null;
5
6      public Toupper( int index ){
7          if ( index == 0 ){
8              value = "value is zero.";
9          }
10     }
11     public String toUpper(){
12         return value.toUpperCase();
13     }
14 }
```

[行12] “this.value” は null の可能性がある

Task List | DTP の指摘事項の詳細 | Bug Reviews | Bug Inf

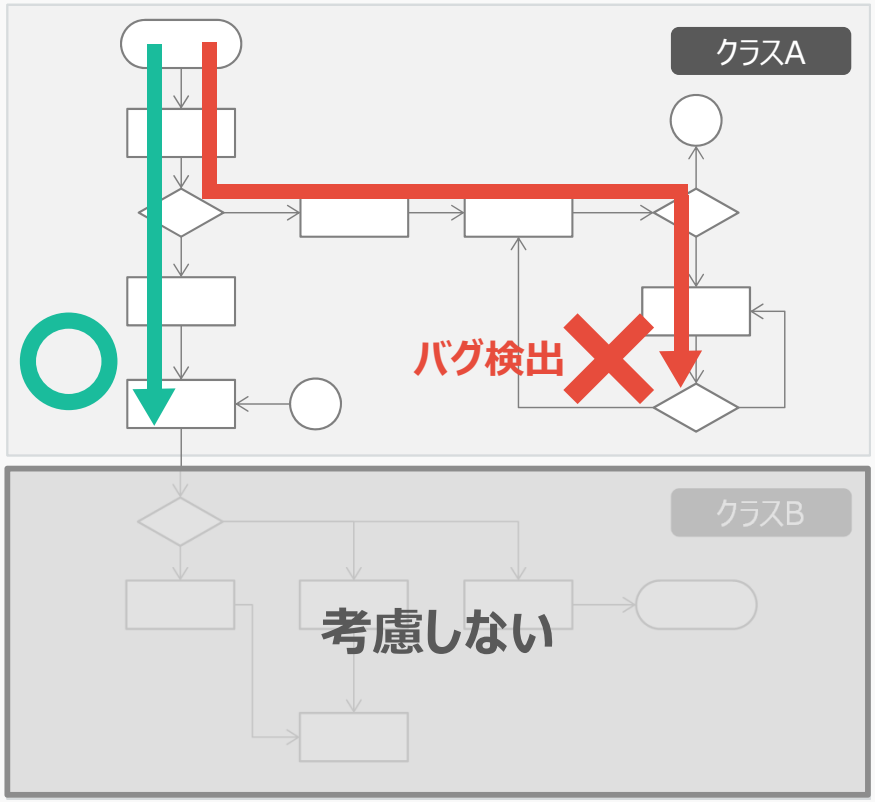
- [行 12] "this.value" は null の可能性がある
  - ToupperCall.java (5): new Toupper(10)
    - Toupper.java (4): public String value = null;
    - Toupper.java (7): if (index == 0){
  - ToupperCall.java (5): Toupper name = new Toupper(10);
  - ToupperCall.java (6): name.toUpper()
    - Toupper.java (12): value.toUpperCase()

Jtestは、ファイル間(クラス間)に
またがった処理の流れを分析します。

SpotBugsとJtestの検出能力の比較

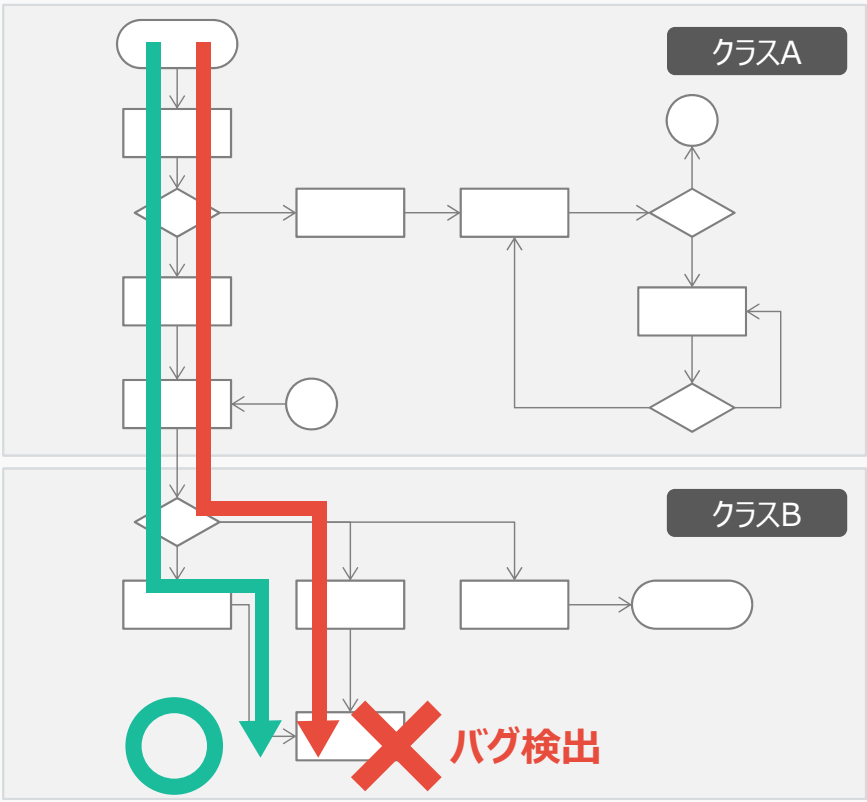
SpotBugs

解析範囲は1ファイル内(1クラス内)に限定



Jtest®

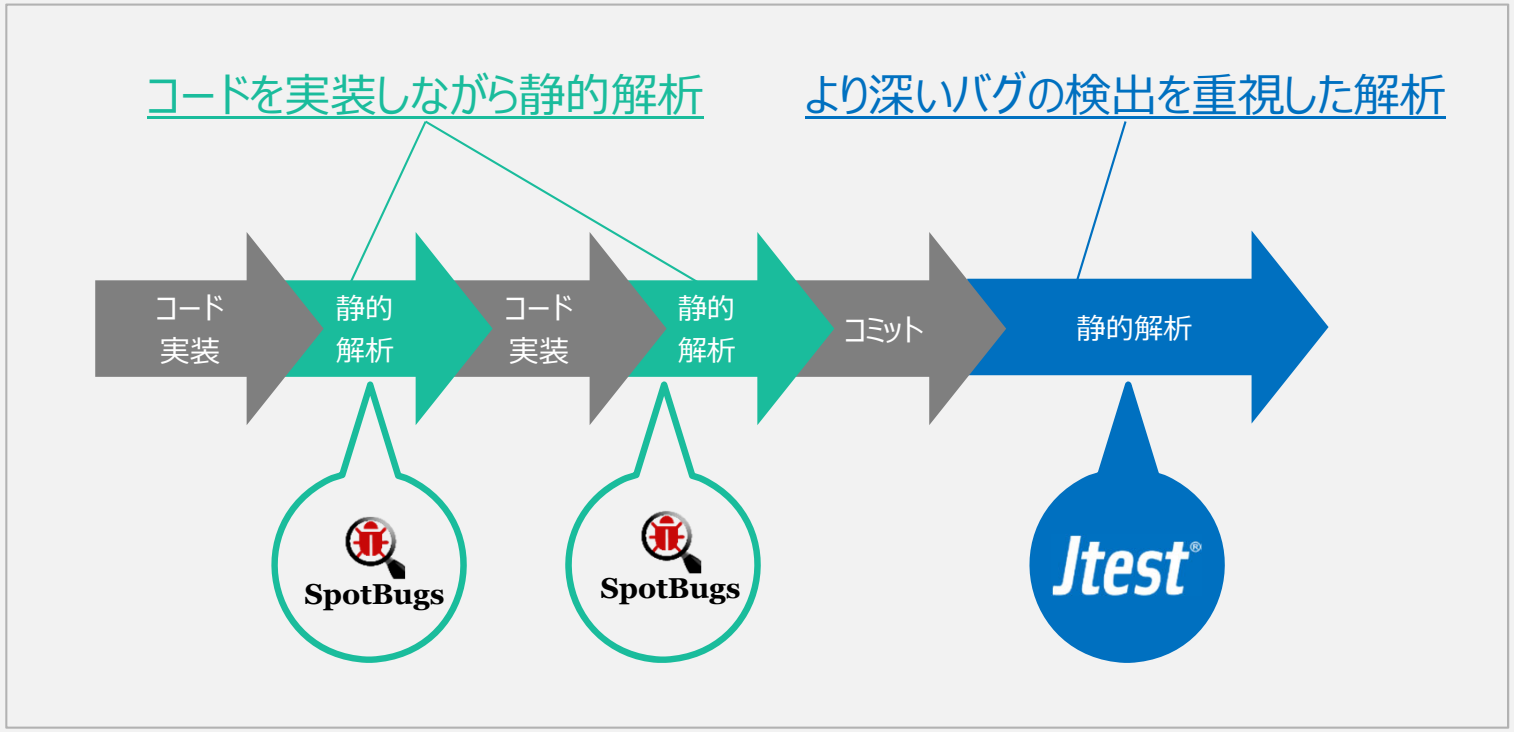
ファイル(クラス)をまたがった処理の流れを分析



ツールの適用範囲を考える

ツールの特性を理解し、効果的に利用する

- SpotBugsは開発者がコードを実装しながら書いたその場でチェックした方が効果的。
- Jtestでは、SpotBugsが検出できない、クラスをまたがった処理の問題やSpotBugsにはない観点の問題の検出ができる。
- 製品コード（実コード）のバグを開発の早い段階で取り除くためには、精度の高い解析ツールを用いるのが良い。
- 解析範囲が広く、精度の高い有償ツールはコミットのタイミングで各開発者のソースコードを集めて解析を行う事で、効果的な運用が出来る。



失敗例1. 期待するバグが検出されない。

自動テストを実行

- 毎日深夜にSpotBugsにて静的解析を実行している。
- 致命的な問題につながる可能性のある実装箇所を検出したい。
- 思ったようにバグが検出されず、機能テスト以降の工程で不具合が検出されてしまう。

ツールの解析精度、
適用範囲に問題がある。

解決方法：

1. 解析能力の高い静的解析ツールを採用する。
   - 検出の精度は利用するツールに依存し、運用の努力では改善できないため、バグ検出能力に優れたツールを採用する。
2. ツールの特性を理解し、適用範囲を検討する。

失敗例2.

ツールにて大量に問題が検出され
修正しきれない。

失敗例2. ツールにて大量に問題が検出され修正しきれない。

自動テストを実行②

- 静的解析ツールによる解析結果が多すぎる。
- ツールのルール一覧を基に検出したいバグを選定してしまうと大量のバグの可能性のある問題が検出される。
- 結局、修正すべきバグが埋もれてしまい修正できない（されない）。

ツールで検出するルールが適切ではない

大量に問題が検出される原因はなにか？

大量に問題が検出される原因

- 解析ツールに搭載されているルール(バグパターン)を基に検出する問題を選定してしまう。

  解析対象のプロジェクトとは無関係に、検出したいルールをピックアップしてしまう。
  例）インターネットや書籍でおすすめされているルールをすべて取り入れる。

- 致命的ではない問題を検出するルールを選定している。

  即座に修正しなくとも動作に影響のない問題を検出している。
  例）インデントのスペース数・タブ数などの問題をツールに検出させている。
  例）未使用コードの検出を毎日ツールで実施している。

適切なルールを選定するためには？

推奨する手順

1. 実際のプロジェクトに対して、静的解析を実行し結果を取得します。
2. 検出した結果をレビューし、不要なルールを取り除きます。

- 直接バグにつながらないルールは外す。

たとえば、インデントのスペース・タブ数に関する問題であれば、IDEの機能で検出できるケースが多いため、自動化されたテストでは検出しない。

また、未使用コードの検出や命名規則に関するものは、頻繁に繰り返す自動化されたテストでは検出しない。リファクタリングやコードレビューのタイミングで検出し、まとめて修正する。

- プロジェクトの決まりや設計通りに実装したら問題となるルールは外す。

ツール上は問題と見なされる記述が、プロジェクトで認められている場合は該当するルールは検出しない。

上記により、開発プロジェクトの開発者数やプロジェクトの予算・工数を考慮した妥当な範囲まで解析結果を絞り込みましょう。

解析結果からルールを選定したJtestの実例

開発プロジェクト　某Webシステム／約20万行／Java

選定前

- ルールの数：107ルール
- 検出件数　：5,100件

選定後

- ルールの数：**22ルール**
- 検出件数　：**262件**

外したルール

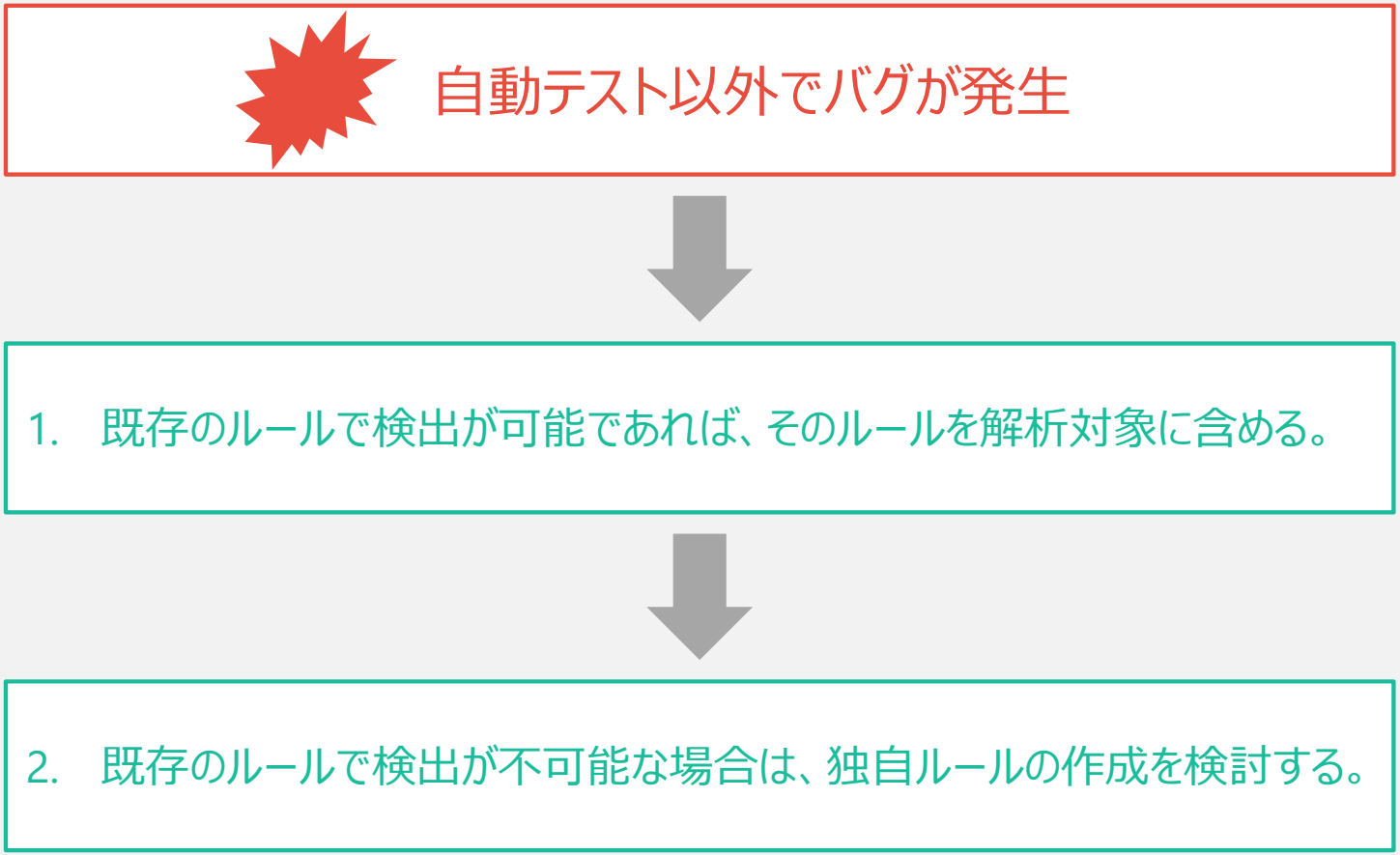
記述作法（2,199件）
規約関連（939件）
命名規則（622件）
Java Doc（622件）
未使用コード（265件）

ルール名	件数
タブではなくスペースを使用する (またはスペースではなくタブを使用する)	1,902
package 名には小文字を使用する	367
型には Javadoc コメントを付ける	286
メソッドには Javadoc コメントと説明を付ける	264
すべてのフィールドを明示的に初期化する	191
1 行の文字数を制限する	145
リテラル定数は使用しない	138
決して読まれないローカル変数の使用を避ける	104
:	

ルール名	件数
特定のクラスの継承を避ける	51
インスタンス フィールドに対する getter および setter メソッドを final で宣言する	43
リソースが割り当て解除されていることを確認する	25
1 つのメソッド内でだけアクセスされる private フィールドを使用しない。ローカル変数に変更する	24
static final 宣言が意図されていた可能性のある static フィールドを検査する	23
非推奨の API を使用しない	17
:	

ルールを選定した後に考慮すること

自動テストの運用を開始し、プロジェクトで検出されたバグに応じてルールを改善します。



1. 既存のルールで検出が可能であれば、そのルールを解析対象に含める。

NullPointerExceptionが発生するコード

Personal.java

```
28  private String getMode( String str ){
29    Integer attValue = hashMap.get( str );
30
31    switch( attValue.intValue() ){
32    case 0:
33      return MEMBER_MODE;
34    case 1:
35      return NO_MEMBER_MODE;
36    }
37    return null;
38  }
39  public void generateCustomer(){
40    BufferedWriter buffWriter = getWriter( "AA" );
41    if ( buffWriter == null || customerList.isEmpty() ) return;
42    try {
43      DataStore dataStore = new DataStore();
44      StringBuilder output = new StringBuilder();
45      for ( int i = 0; i < customerList.size(); i++ ){
46        String id = customerList.get( i );
47        output.append( id );
48        String customerMode = getMode( id );
49        if ( customerMode.equals( MEMBER_MODE ) ){
50          output.append( "," ).append( discountMember );
51        } else {
52          output.append( "," ).append( noDiscount );
53        }
54        dataStore.storeData( id );
55      }
56
57      buffWriter.write( output.toString() );
```

nullを返却する可能性のあるメソッド(getModeメソッド)

getModeメソッドの戻り値を
そのまま利用(customerMode)

このケースにおいて、NullPointerExceptionの発生を防止するためにはどうすればよいでしょうか？

1. 既存のルールで検出が可能であれば、そのルールを解析対象に含める。

このケースにおいて、NullPointerExceptionの発生を防止するためにはどうすればよいでしょうか？

- NullPointerExceptionが発生した根本的な原因、解決方法を調査します。

問題のあるコード

```
41    if ( buffWriter == null || customerList.isEmpty() ) return;
42    try {
43      DataStore dataStore = new DataStore();
44      StringBuilder output = new StringBuilder();
45      for ( int i = 0; i < customerList.size(); i++ ){
46        String id = customerList.get( i );
47        output.append( id );
48        String customerMode = getMode( id );
49        if ( customerMode.equals( MEMBER_MODE ) ){
50          output.append( "," ).append( discountMember );
51        } else {
```

```
if ( customerMode.equals( MEMBER_MODE ) ){
```

String.equals(リテラル)

修正したコード

```
41    if ( buffWriter == null || customerList.isEmpty() ) return;
42    try {
43      DataStore dataStore = new DataStore();
44      StringBuilder output = new StringBuilder();
45      for ( int i = 0; i < customerList.size(); i++ ){
46        String id = customerList.get( i );
47        output.append( id );
48        String customerMode = getMode( id );
49        if ( MEMBER_MODE.equals( customerMode ) ){
50          output.append( "," ).append( discountMember );
51        } else {
```

```
if ( MEMBER_MODE.equals( customerMode ) ){
```

リテラル.equals(String)

equalsの呼び出しを変更することで、問題を解決できる。

- 根本的な原因や解決方法から、合致する既存ルールを解析対象に追加します。

equalsの呼び出しに関するコーディング規約のルール※を解析ルールに追加します。

※ Jtestの場合は「String.equals(constant) または String.equalsIgnoreCase(constant) を呼び出してはいけない」を追加します。

2. 既存のルールで検出が不可能な場合は、独自ルールの作成を検討する。

検出された問題に関して、既存のルールでは検出が出来ない場合、ルールの作成を検討します。

ツールごとのルール作成方法、注意点

ツール名	作成方法	注意点
SpotBugs	Javaでルールを実装する。	実装にはバイトコードを理解する必要がある。
CheckStyle		Javaの言語仕様を正しく理解する必要がある。 実装したルール自体にバグが混入する可能性がある。
PMD		
Jtest	独自のGUIを用いてルールを実装する。	GUIの操作方法を習得する必要がある。 言語仕様を細かく理解しなくても良い。

2. 既存のルールで検出が不可能な場合は、独自ルールの作成を検討する。

独自ルール作成の例

- 発生した問題と原因

エラー発生時にデータがロールバックされなかった。

原因：

顧客データの保存でトランザクション処理が行われていなかった。データの保存は以下のクラスを利用しており、トランザクション処理を行う場合、storeDataの呼び出し前にstartTransactionを呼び出す必要があった。

DataStore
- con: Connection
+ startTransaction(): boolean + storeData(str: String): boolean + rollBackTransaction(): boolean + endTransaction(): boolean

- 対策

storeData呼び出しの前にstartTransactionを呼び出していない実装を検出し、
類似バグの発生を防止する。

2. 既存のルールで検出が不可能な場合は、独自ルールの作成を検討する。

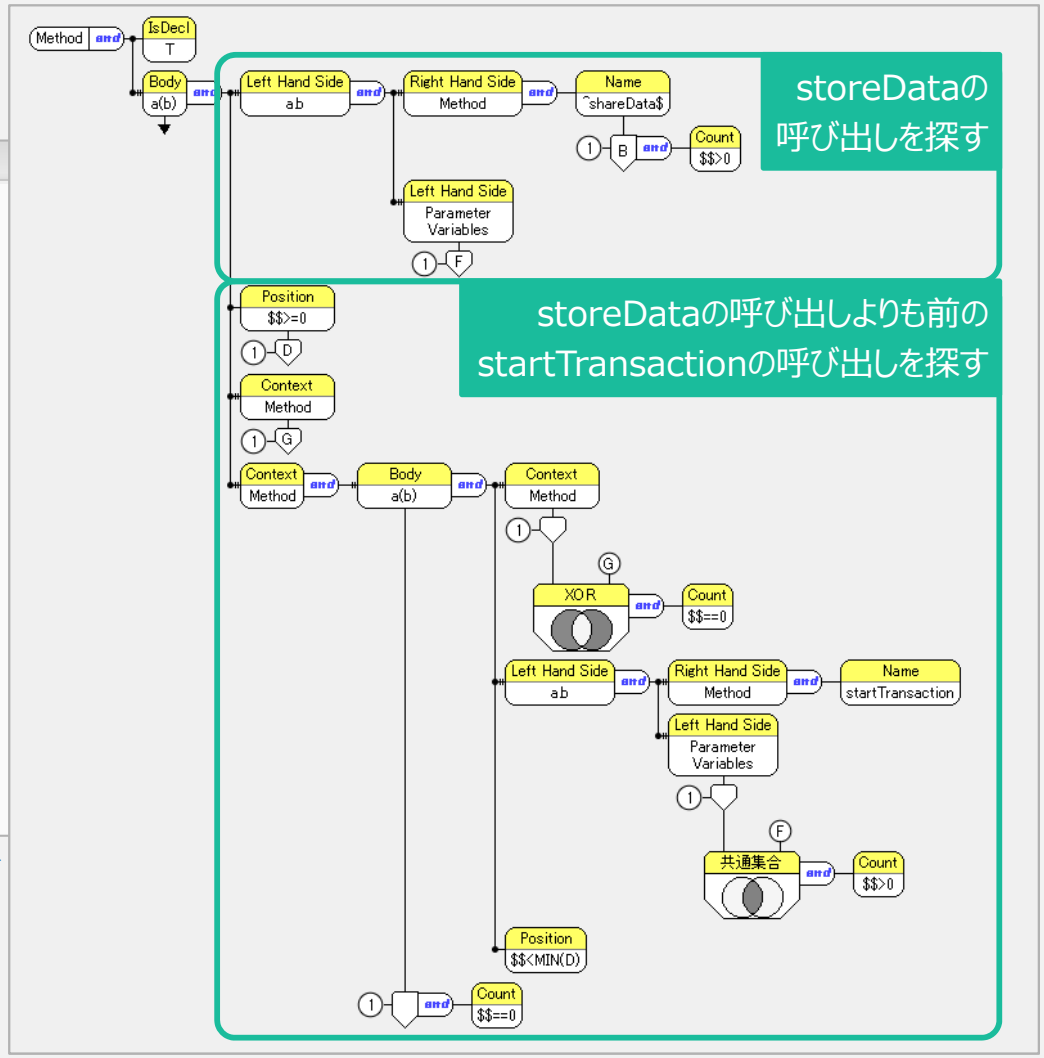
独自ルール作成の例

Personal.java

```
39  public void generateCustomer(){
40    BufferedWriter buffWriter = getWriter( "AA" );
41    if ( buffWriter == null || customerList.isEmpty() ) return;
42    try {
43      DataStore dataStore = new DataStore();
44      StringBuilder output = new StringBuilder();
45      for ( int i = 0; i < customerList.size(); i++ ){
46        String id = customerList.get( i );
47        output.append( id );
48        String customerMode = getMode( id );
49        if ( customerMode.equals( MEMBER_MODE ) ){
50          output.append( "," ).append( discountMember );
51        } else {
52          output.append( "," ).append( noDiscount );
53        }
54        dataStore.storeData( id );
```

storeData() の前にstartTransactionが呼び出されていない実装を見つけたい

Jtest® のルール作成画面



プロジェクト固有のバグに関しても、独自のルールを作成し
自動化したテストを行うことで再発を防止します。

大量に問題が検出される別の例

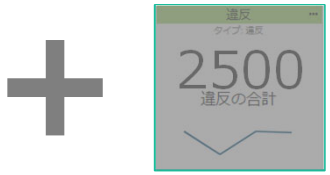
保守開発や開発の途中でツールを導入したら大量に問題が検出された

Jtest® の場合…

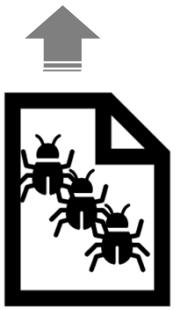
既存のコードにツールを適用すると
大量の問題が検出

この解析をベースラインに設定

ベースライン以降に混入した違反のみレポート



ベースライン以前に検出した問題は表示しない。



ソースコードの修正により
3件の問題が新たに混入

対策

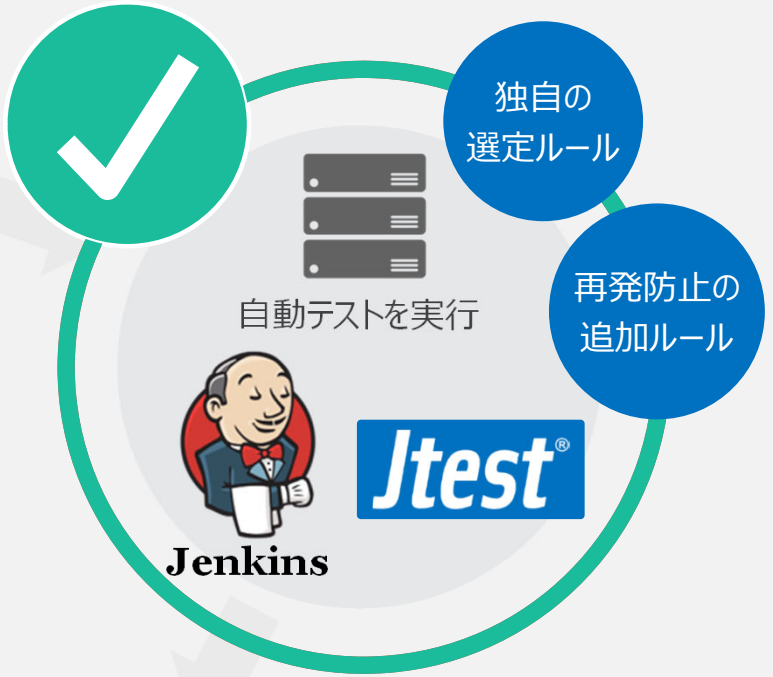
- ソースコードの追加、修正した事によって発生した問題にだけ対応する

すでに可動しているシステムであれば、潜在バグになりうるものの、問題は無いと判断し、ソースコードの変更によって新たに発生した問題にだけ対応する。

失敗例2. ツールにて大量に問題が検出され修正しきれない。

自動テストを実行②

- 静的解析ツールによる解析結果が多すぎる。
- ツールのルール一覧を基に検出したいバグを選定してしまうと大量のバグの可能性のある問題が検出される。
- 結局、修正すべきバグが埋もれてしまい修正できない（されない）。



ツールで検出するルールが適切ではない

解決方法：

1. 静的解析の結果を基に、ルールを選定する。
2. 選定したルールは検出した問題に応じて改善する。

＋

ソースコードの追加・修正によって新たに発生した問題だけ対応する

失敗例3.

検出された問題が修正されない。
（放置される）

失敗例3. 検出された問題が修正されない。（放置される）

検出した問題の確認、修正

- 解析した結果が放置され修正されない。
- 自動化テストに問題の確認および修正の仕組みがないケースが多い。
- 問題が放置され累積しだすと将来的には大きな技術的負債になる。

問題の確認・修正が非効率であり、手間がかかる

検出された問題が放置される原因はなにか？

検出された問題が放置される原因

- 検出された問題の数が多い、または工数がなく対応しきれない。（失敗例2の件に関連）
- 検出した問題の共有方法に問題がある。

バグの検出を自動化したはいいが、確認や修正などの検出後のプロセスが弱い。
例）検出された問題がメールやファイルで展開される。

対策

- テスト自動化の一部として、検出された問題の共有およびアサインを自動的に行う。

結果の集計やアサインが人手で行ってしまうと、自動化しているメリットが半減する。
静的解析でバグを検出したらすぐに結果を共有し、アサインを行う。

今回の開発プロジェクトでは

- 結果の確認がJenkins上、アサインがExcelを利用しており、共有およびアサインの方法が効率的ではありません。
- 開発者がより内容を確認しやすい、かつアサインされる仕組みを取り入れます。
- より効果的な共有方法とアサイン方法をご説明します。

検出された問題をテスト自動化の中で共有する方法

自動的に検出された問題を共有する方法の例

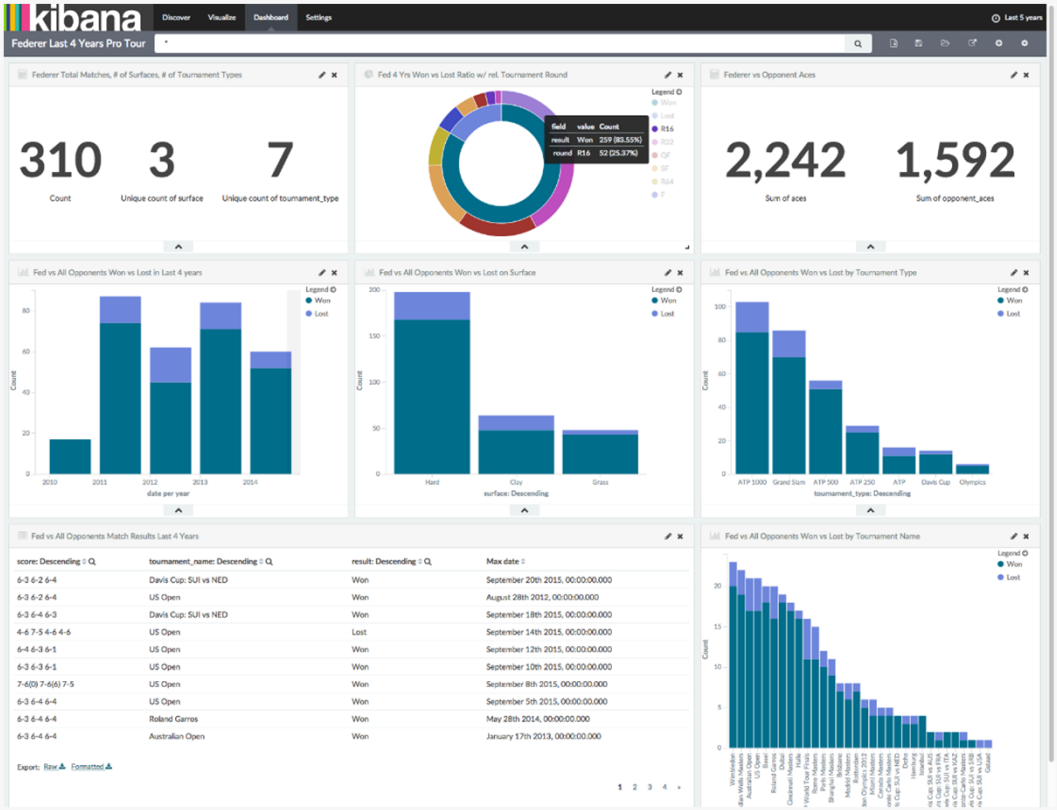
共有方法		特徴
解析結果をファイルやメールで共有	△	● JenkinsなどのCIツールや自作スクリプトで対応しやすい。
課題管理システム等で共有	○	● プロジェクトの情報として一元管理しやすい。 例）Redmine, Bugzilla, AccuRevなど
開発環境上で共有	○	● 使い慣れた環境でソースコードと共に確認できる。 例）Eclipse, IntelliJ, NetBeansの各Pluginなど

データ共有の現在の動向

ダッシュボードツール

- さまざまなデータを集約して算出した指標を統合された画面に数値やグラフで表示します。
- 主に、データの可視化や分析に利用されます。
- ソフトウェア開発における活用例
  - バグの統計情報
  - 担当者ごとの担当タスク件数
  - コード行数などのメトリクスの推移
  - テストの失敗数の推移
- ソフトウェア開発で利用した場合、バグの情報だけではなく、プロジェクト全体の統計情報を共有できるようになります。

Kibana + Elastichsearchの例

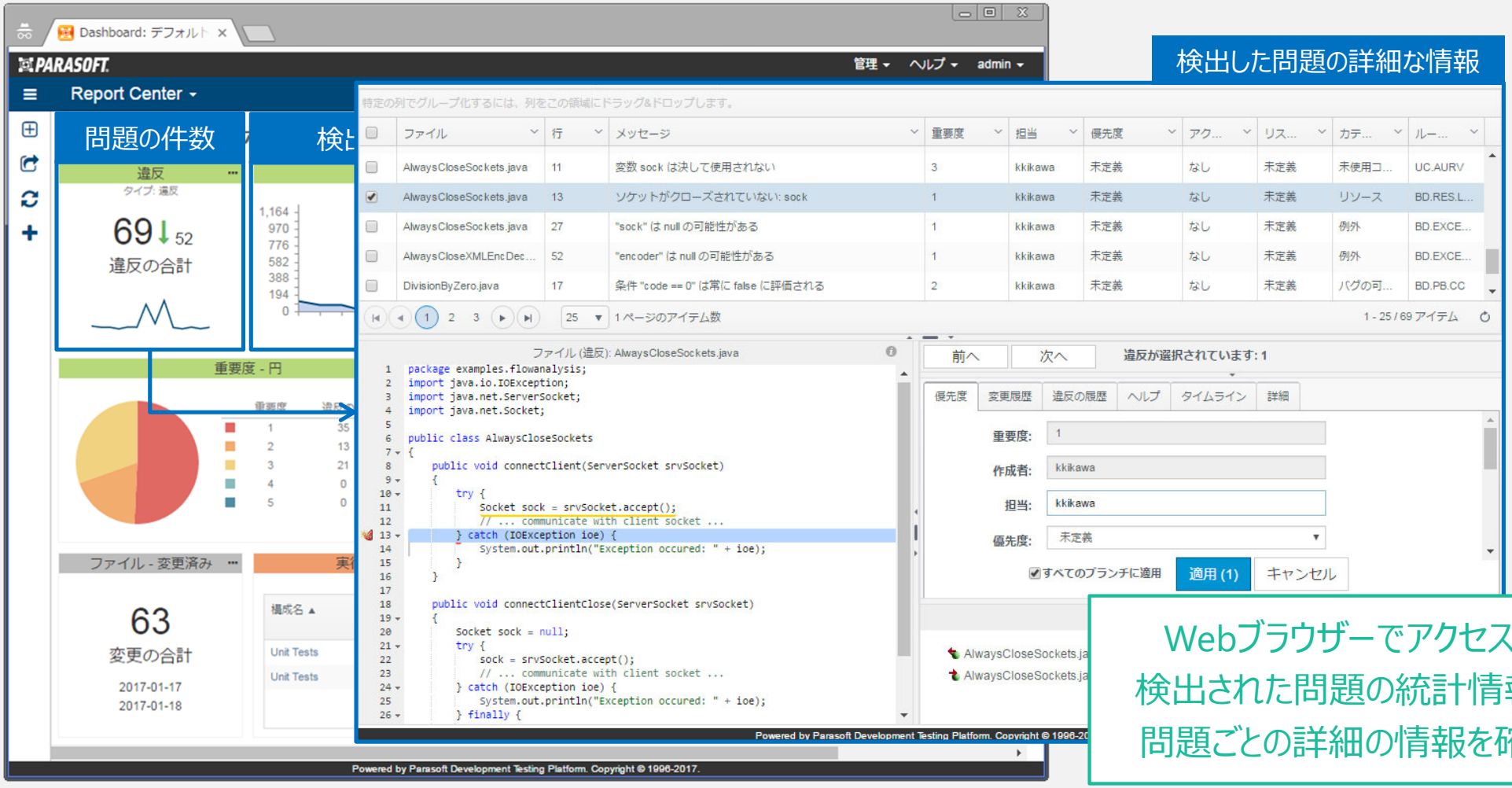


https://www.elastic.co/blog/building-dashboards-using-data-from-the-federer-&-djokovic-tennis-rivalry

検出した問題を共有する

PARASOFT® DTP
Development Testing Platform

- 自動化テストの結果を可視化するダッシュボードツール
- 収集データを分析し、独自指標をグラフ化

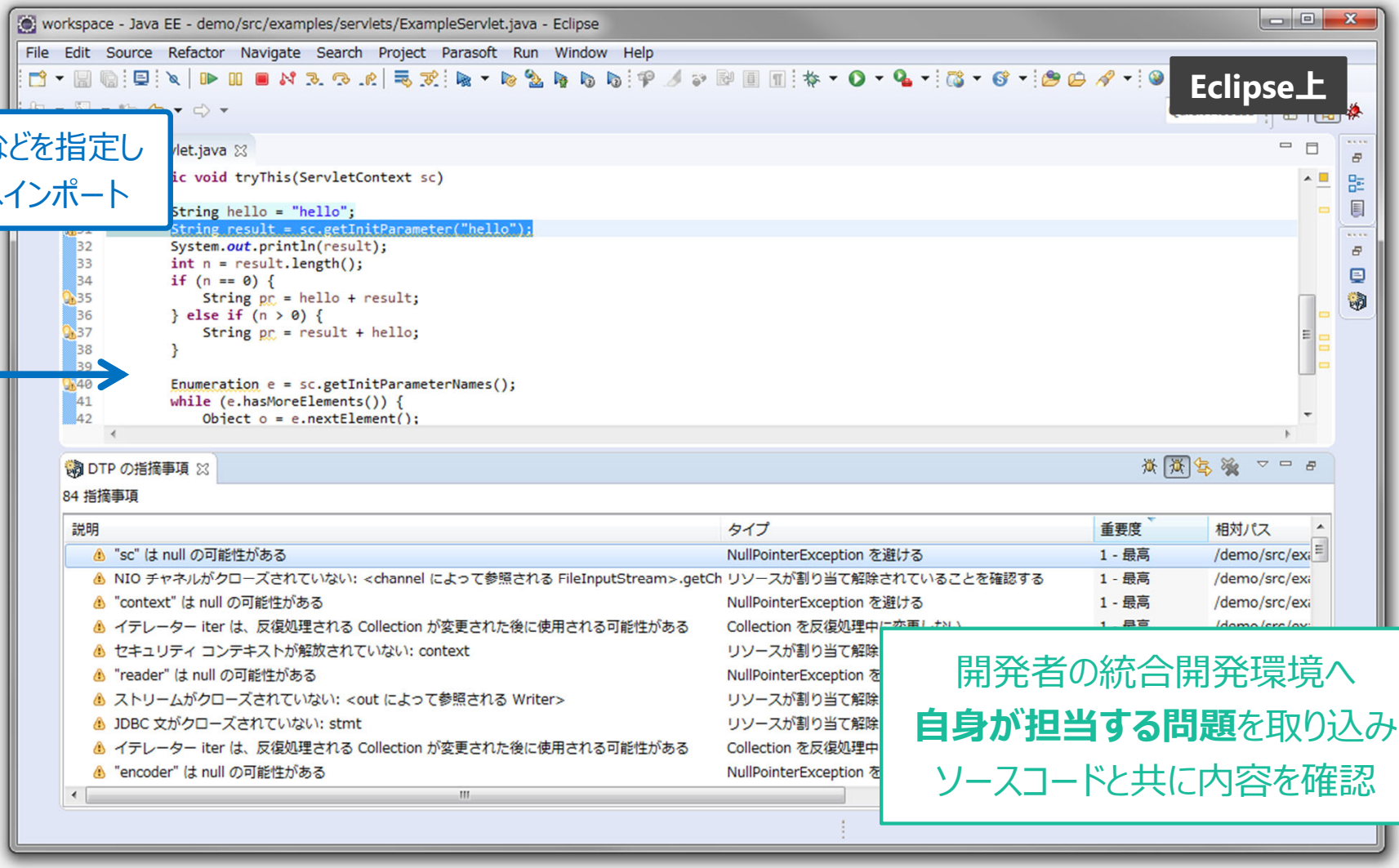


Webブラウザーでアクセスし
検出された問題の統計情報や
問題ごとの詳細の情報を確認

検出した問題を共有する

PARASOFT® DTP
Development Testing Platform

重要度、担当などを指定し
ローカル環境へインポート



開発者の統合開発環境へ
自身が担当する問題を取り込み
ソースコードと共に内容を確認

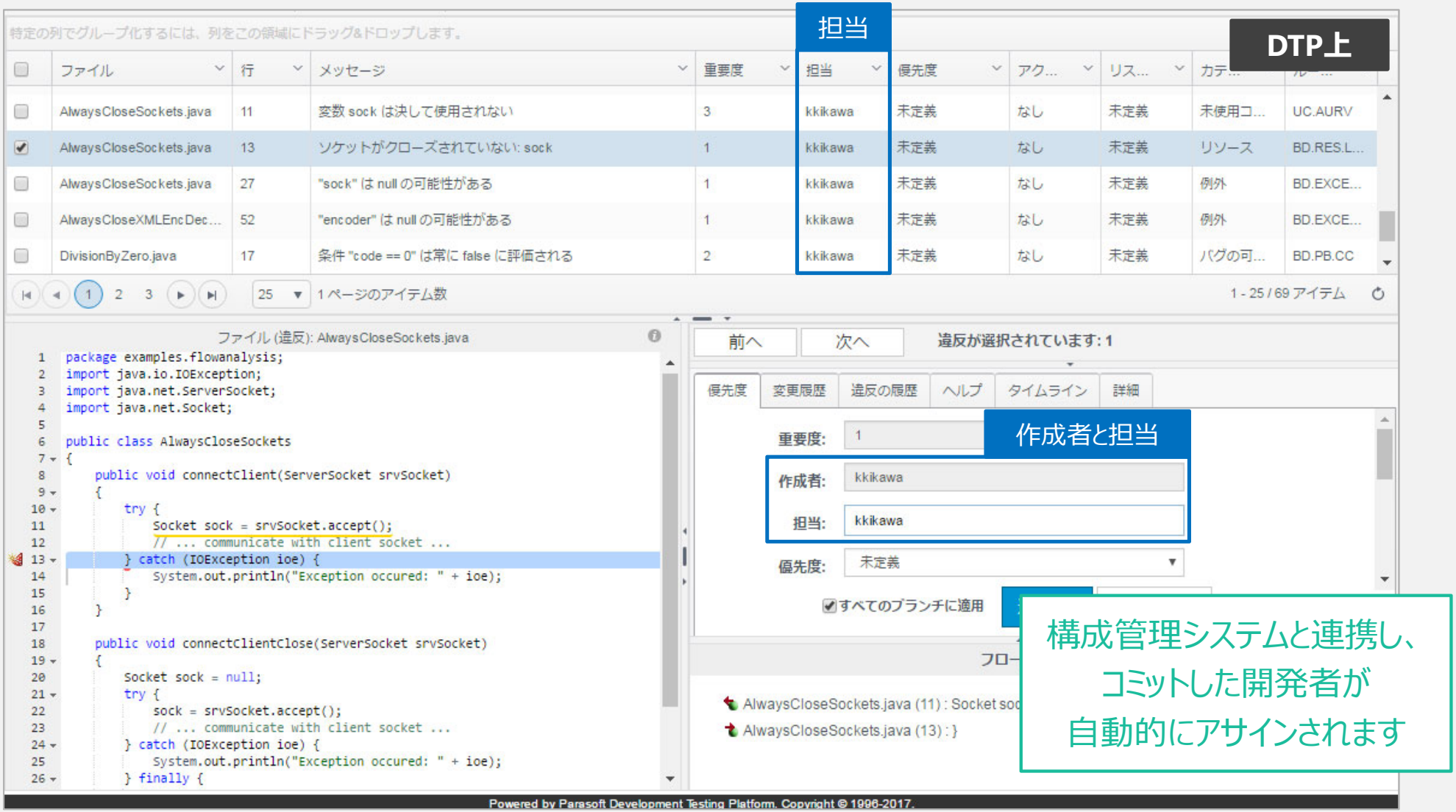
検出された問題をテスト自動化の中でアサインする方法

自動的に検出された問題をアサインする方法の例

アサイン方法		特徴
あらかじめ決められた担当者へ機械的にアサインする	△	● 未アサインは減るものの、アーキテクチャや担当者の変更の度にアサインの仕組みを変更する必要がある。 ● 一部の担当者へアサインが集中する可能性がある。
構成管理のコミット情報から担当者をアサインする	○	● コードを直近で修正した人をアサインするため、一次調査が進みやすい。 ● 機械的にアサインできる上にメンテナンスコストが安い。

検出した問題をアサインする

PARASOFT® DTP
Development Testing Platform



失敗例3. 検出された問題が修正されない。（放置される）

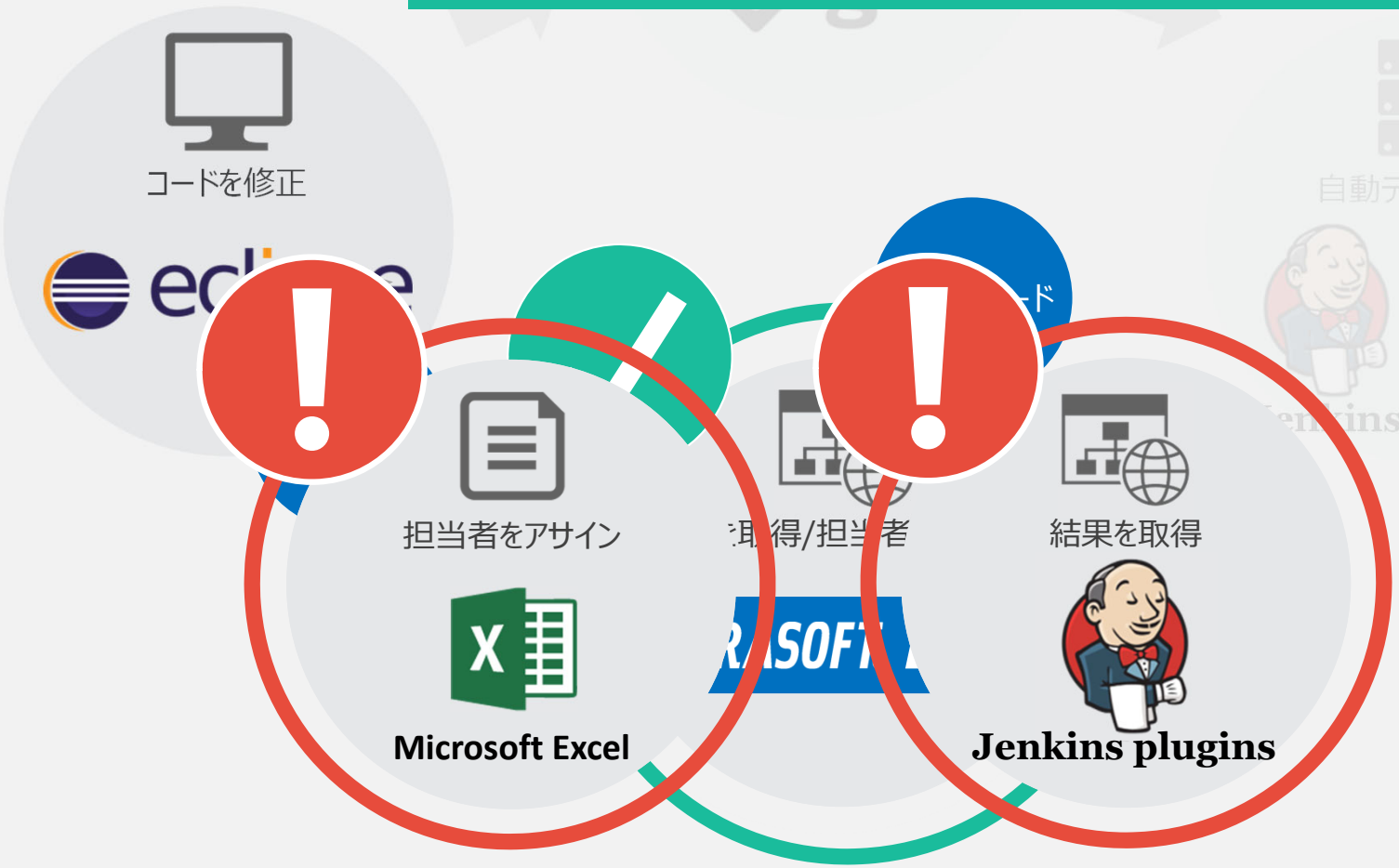
検出した問題の確認、修正

- 解析した結果が放置され修正されない。
- 自動化テストに問題の確認および修正の仕組みがないケースが多い。
- 問題が放置され累積しだすと将来的には大きな技術的負債になる。

問題の確認・修正が非効率であり、手間がかかる

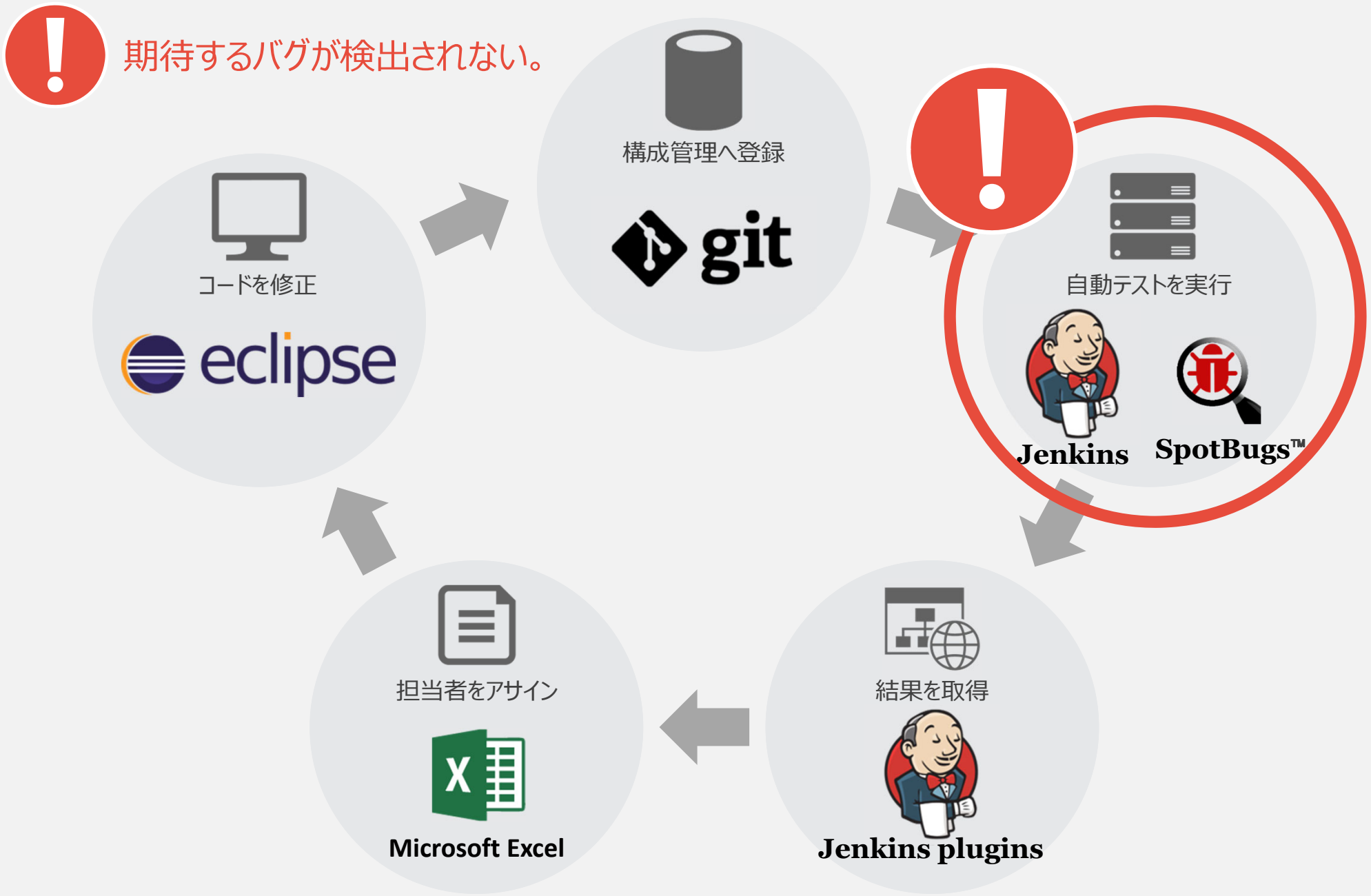
失敗例3. 検出された問題が修正されない。（放置される）

解決方法：
問題を検出した時点で作業担当者をアサインし、
さらに開発者の環境で容易に結果を閲覧できるようにする。

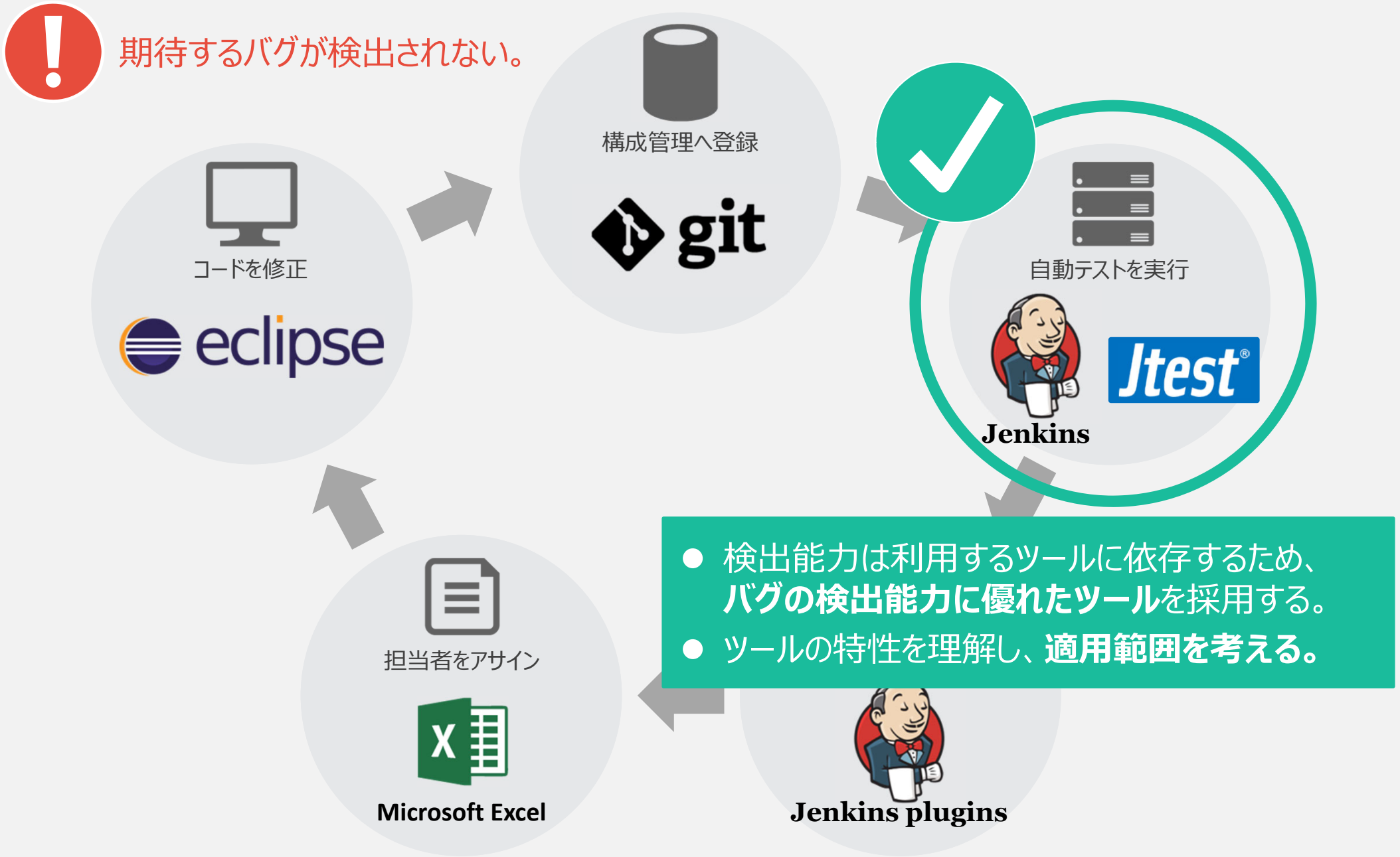


まとめ
ソフトウェアのバグ検出における テスト自動化のポイント

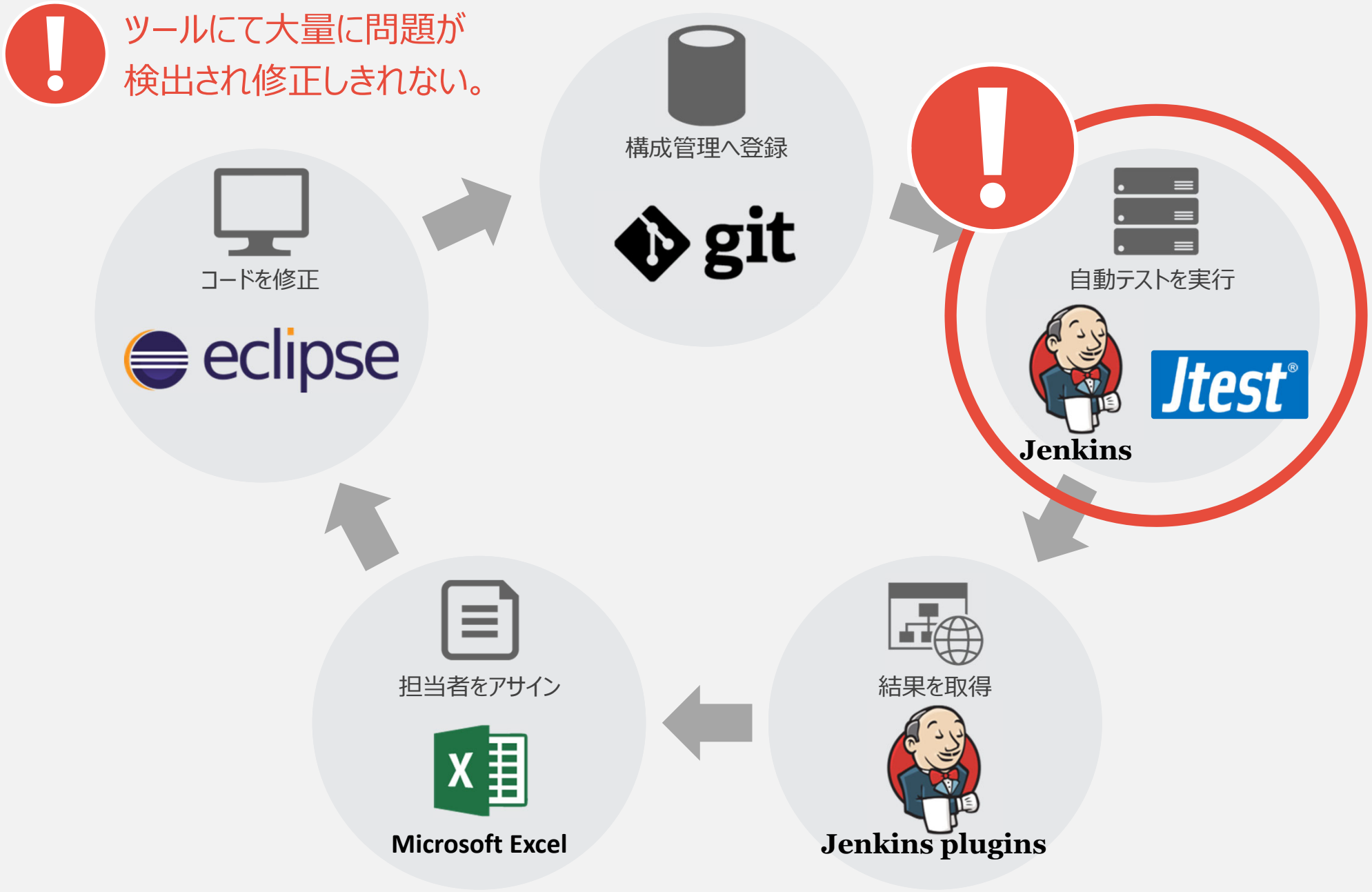
ソフトウェアのバグ検出におけるテスト自動化のポイント



ソフトウェアのバグ検出におけるテスト自動化のポイント

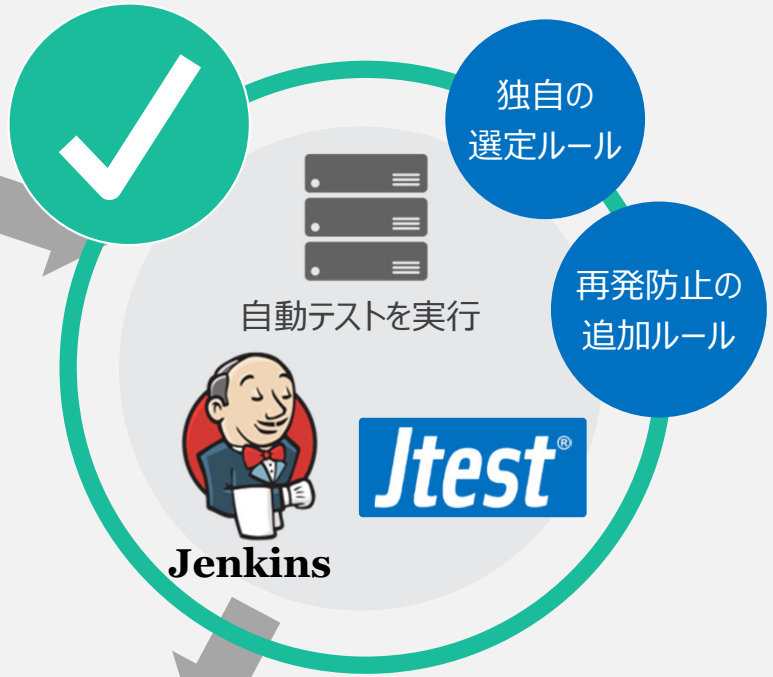


ソフトウェアのバグ検出におけるテスト自動化のポイント



ソフトウェアのバグ検出におけるテスト自動化のポイント

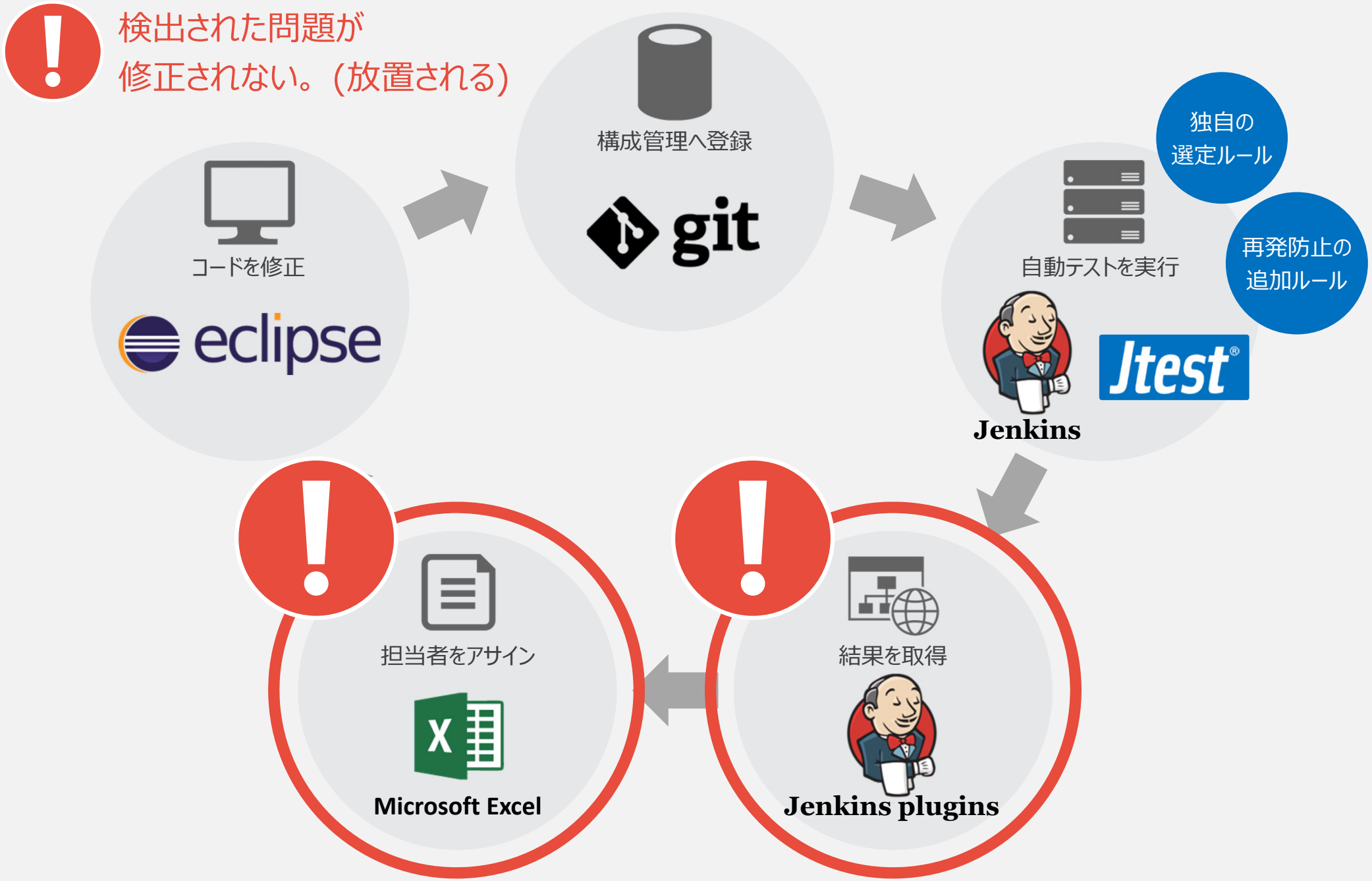
ツールにて大量に問題が
検出され修正しきれない。



- **解析内容をプロジェクトに合わせて設定**する。
- 静的解析の結果から解析ルールを選定し、
  発生した問題に応じて**選定ルールを改善する。**
- 保守開発の場合はソースの追加・修正によって
  発生した**新規の問題だけ対応**する。

Jenkins plugins

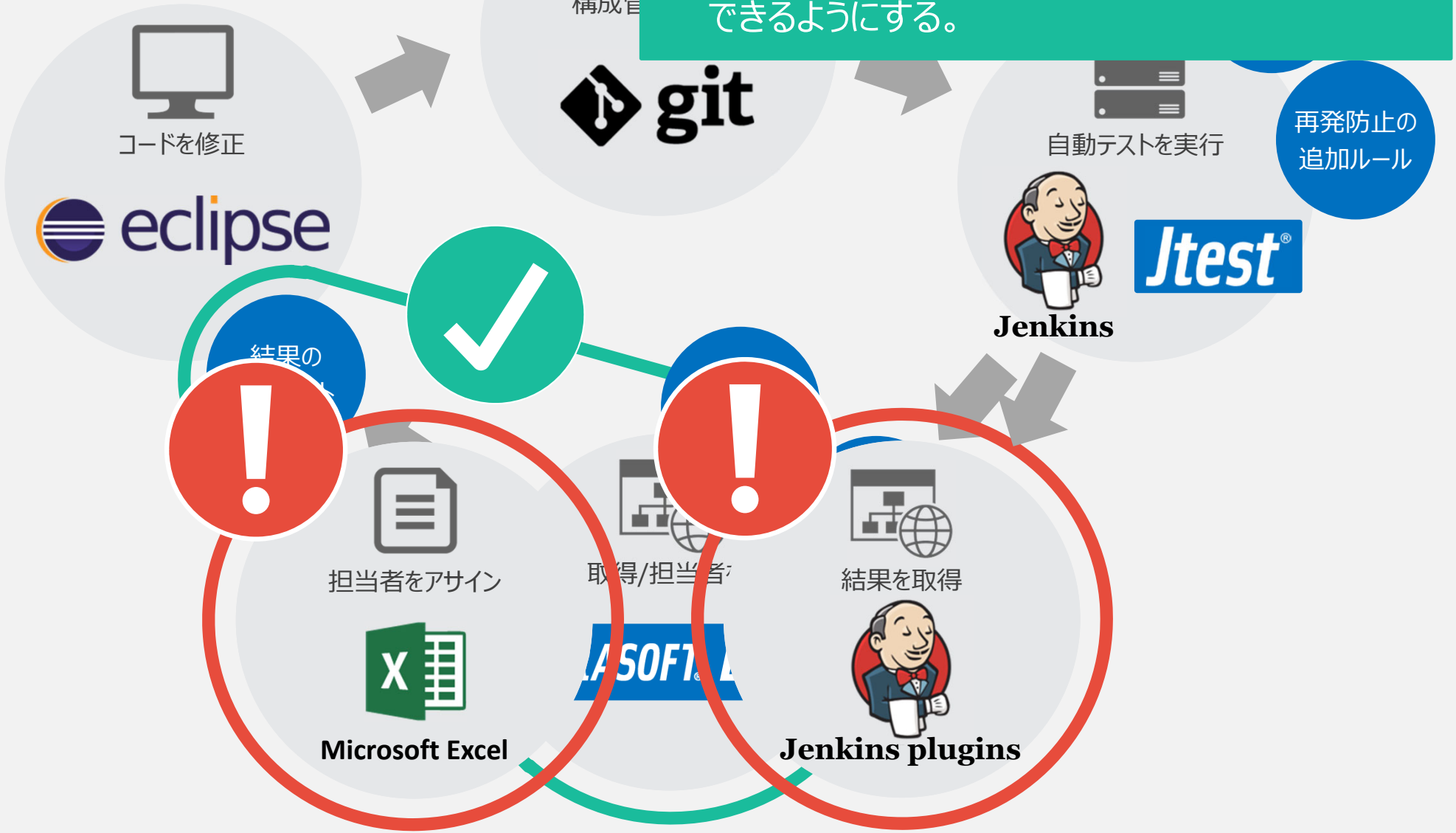
ソフトウェアのバグ検出におけるテスト自動化のポイント



ソフトウェアのバグ検出におけるテスト自動化のポイント

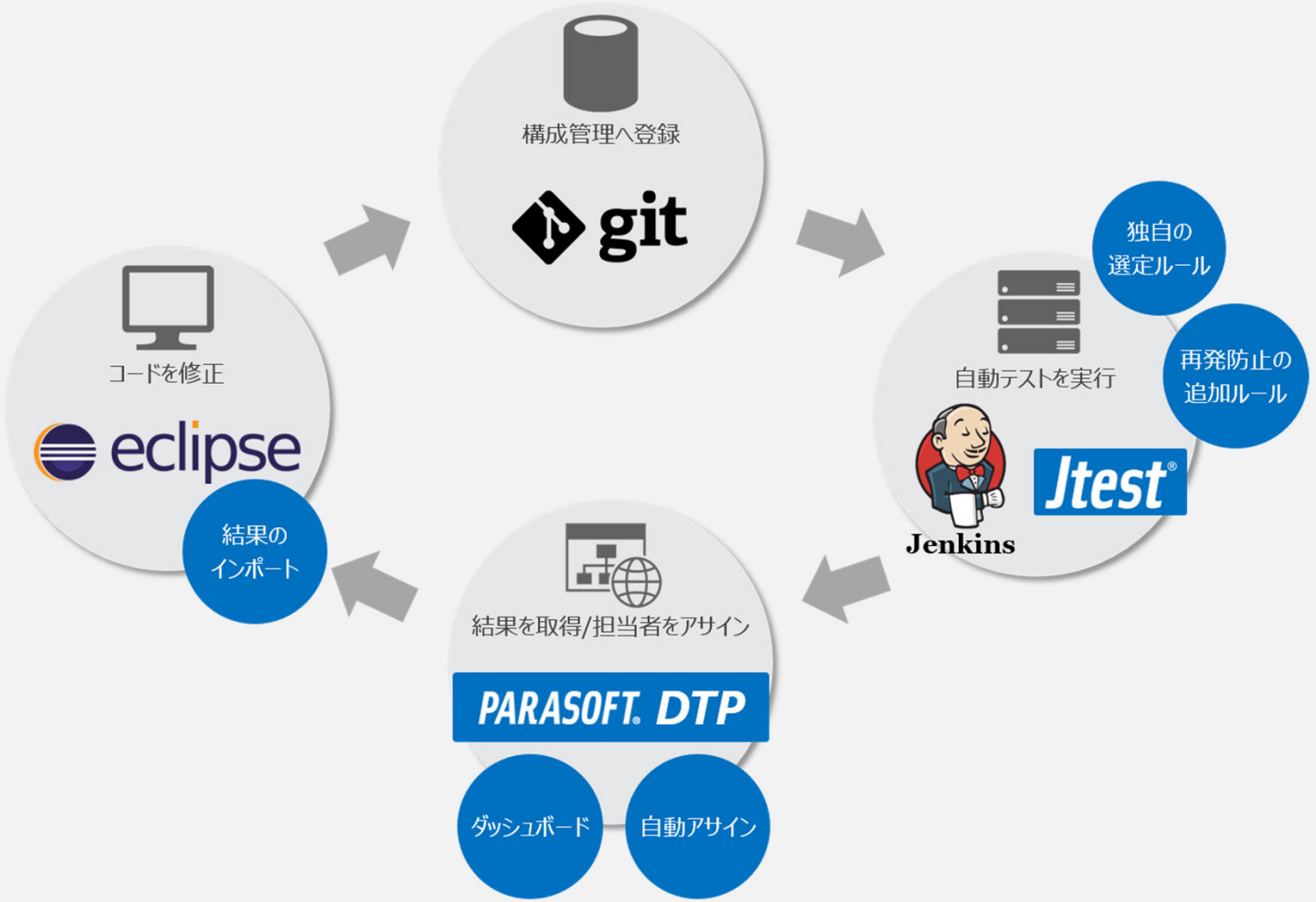
検出された問題が
修正されない。(放置される)

- **担当者のアサインを自動化**する。
- **開発者環境でソースコードと共に結果を閲覧**できるようにする。



ソフトウェアのバグ検出におけるテスト自動化のポイント

テスト自動化におけるバグの検出では、実プロジェクトにあったツールやルールの選定、
開発者がバグ修正に取り組みやすい仕組みを構築することが大切です。



お問い合わせ先

テクマトリックス株式会社
システムエンジニアリング事業部
ソフトウェアエンジニアリング営業部

☎ 03-4405-7853
03-6436-3553
se-info@techmatrix.co.jp
https://www.techmatrix.co.jp/product/jtest/