SYNOPSYS®
Silicon to Software™

効率的にシフトレフトをするための静的解析

日本シノプシス合同会社
シニアセールスエンジニア
伊藤俊廷
2019.07.18

目次

- アジャイル開発の事例紹介
- シフトレフトとテスト自動化
- 静的解析ツールCoverity
- 静的解析導入のポイント
- まとめ

アジャイル開発の事例紹介

経験プロジェクトの概要

目的	Webアプリフレームワーク製品利用時の、手本となるサンプル（**テンプレートアプリ**）を提供し、Webアプリの開発手法や製品の学習に役立てる
開発ソフトウェア	Webアプリフレームワーク製品に付属するテンプレートアプリを、アーキテクチャパターン（DBへのCRUDやSOAP接続など）ごとに計10個提供
最終的な成果物	• 本番相当の環境、ローカル環境の双方で動作する**アプリケーション** • 手本となるアプリケーションの**ソースコード、ユニットテストコード** • アーキテクチャの概要、開発の手順を解説した**ユーザガイド**
チーム規模	約5人の開発者
イテレーション期間、回数	２週間を約10イテレーション実施
取り入れたプラクティス	プランニングポーカー、バーンダウンチャート、ユニットテスト自動化、統合テスト自動化（非UIテスト）

アジャイル開発で気づいた良かった点

- どこまで作り込むか曖昧な成果物への要求事項が、イテレーションを回すにつれて固まる
  - 1つ目のアプリを企画側がレビューしたことにより、ベースとなる要求レベルを確認できた
    プロトタイプ開発に似た効果が得られ、QCDSのQualityとScopeの確認ができた
- 各工程でやるべきことが早いうちに把握できる
  - エンジニアとして「今まで経験した工程は基本設計と統合テストです」ということが起きない
- プランニングポーカー、バーンダウンチャートによる計画見積もり精度が向上する
  - 人間は絶対的に見積もるのが苦手、ガンチャート管理はつらい
- テスト自動化の経験が得られる
- プロセスを考えながら改善するのは楽しい

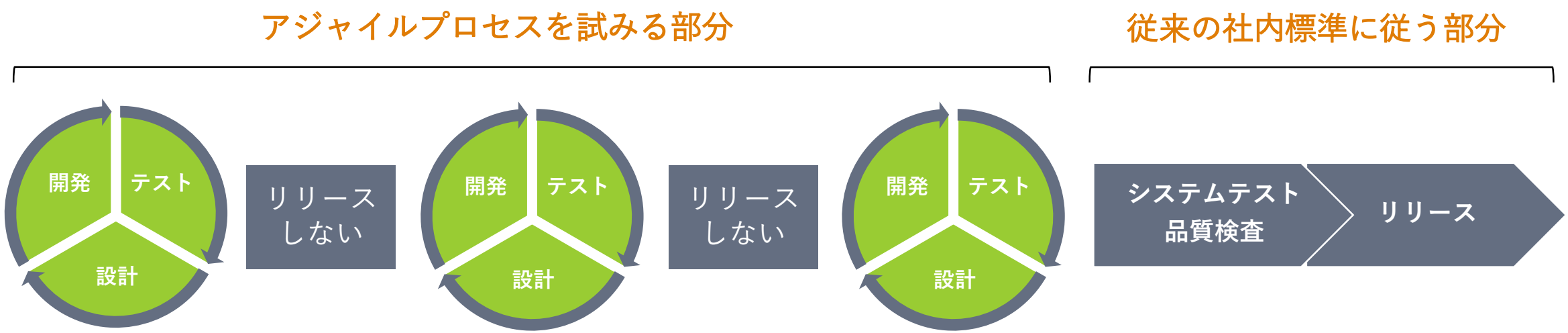
アジャイル開発で気づいた問題点

- 1日8時間でタスクを見積もるから結局残業をする
  - 打ち合わせや他のタスクもあるので、残業しないためには5時間を前提にするべき
- バーンダウンチャートのメンテに時間がかかる
  - Excelだったため、ついついカスタマイズしたり、色々な自動計算を入れたりしてしまう
- 製品リリースの予定された計画が大前提のため、最終的なDeliveryは結局調整できない
  - 部署内の一部取り組みだったので、仕方がない
- イテレーションが進むにつれて成果物の規模が増えるため、共通的なレビュー指摘に対する修正コストは、最終的にはウォーターフォールと変わらなくなる
  - それでも一気に作ってから一度に全部指摘されるよりはマシ
- Wordベースのユーザガイドを2週間に一度のリリースするは効率が悪い
  - 特にキャプチャの差し替え、図表の修正はコストがかかるため、仕上げはイテレーション外でするべき

WF文化の組織における現実的なアジャイル開発

いきなり完全なアジャイルプロセスでの開発するのは難しいため、

部分的に変えていく



アジャイルトライアルあるある

組織やプロセスを変えるのは時間がかかる

- 部分的なアジャイルプロセスになり、最終的な判定は過去のプロセスにしたがう
  - なかなか変えるのが難しい部分
- JIRA、Redmine、Trelloにログインしてくれない上司
  - 「状況はJIRAのこのチケットに書きました」、「その指示はチケットにも記載をお願いします」
- 請負契約のため、物理的に分離されたチームメンバー
  - SIerあるある。 意識的な対面での会話とチャットツールを最大限に活用
- 結局調整できないDeliveryとScope
  - 経営レベルを巻き込んで、製品、サービスの対外的な計画自体もアジャイルにする合意が必要

重要なテスト工程であるソースコードのレビュー

ソースコードレビューは時間、工数がかかる

→機械的にチェックできないレビューに焦点を置きたい

有識者によるレビューは有用である一方、レビュー品質は均一でない

→初期に仕込んだ問題を後のイテレーションで指摘されることも多々ある

人力で時間をかけてレビューしたい対象

- DB排他制御（楽観ロック、悲観ロック）の方式
- アプリケーションフレームワークのログ部品の効果的な使い方
- ローカル環境でテストするためのモックスタブの開発方法
- 正しいエラーのハンドリング方法

機械的に均一にすぐに指摘してほしい対象

- Nullチェック忘れ
- リソースリーク
- デッドコード
- SQLインジェクション

シフトレフトとテスト自動化

本日お伝えしたいこと

開発ライフサイクルの早期から品質担保するには、ソースコード静的解析が重要

クイズ：どこに問題があるでしょうか？（Java 8）

```
7   public class RunBadDBConnection {
8
9       public static void main(String[] args) {
10
11          Connection connection = null;
12          try {
13              connection = DriverManager.getConnection(
14                      "jdbc:oracle:thin:@127.0.0.1:1521:TESTDB", "USER01", "PASSWORD"
15                      );
16          } catch (SQLException e) {
17              e.printStackTrace();
18              return;
19          }
20
21          Statement statement = null;
22          ResultSet resultSet = null;
23
24          try {
25              statement = connection.createStatement();
26              resultSet = statement.executeQuery("select * from config");
27              // Do something
28              resultSet.close();
29              statement.close();
30              connection.close();
31          } catch (SQLException e) {
32              e.printStackTrace();
33          }
34
35      }
36  }
```

ここで例外が発生した場合、インスタンスconnectionはクローズされない

解答：例外パスも含め確実にリソースを開放する(CWE-404)

```
21          try {
22              statement = connection.createStatement();
23              resultSet = statement.executeQuery("select * from config");
24              // Do something
25          } catch (SQLException e) {
26              e.printStackTrace();
27          } finally {
28              try {
29                  if (resultSet != null) {
30                      resultSet.close();
31                  }
32              } catch (SQLException e) {
33                  e.printStackTrace();
34              } finally {
35                  try {
36                      if (statement != null) {
37                          statement.close();
38                      }
39                  } catch (SQLException e) {
40                      e.printStackTrace();
41                  } finally {
42                      try {
43                          if (connection != null) {
44                              connection.close();
45                          }
46                      } catch (SQLException e) {
47                          e.printStackTrace();
48                      }
49                  }
50              }
51          }
52
53      }
54  }
```


参考URL：”FIO04-J. 不要になったリソースは解放する”. JPCERT CC. https://www.jpcert.or.jp/java-rules/fio04-j.html, (2019-06-01)

解答：try-with-resourcesを活用した書き方

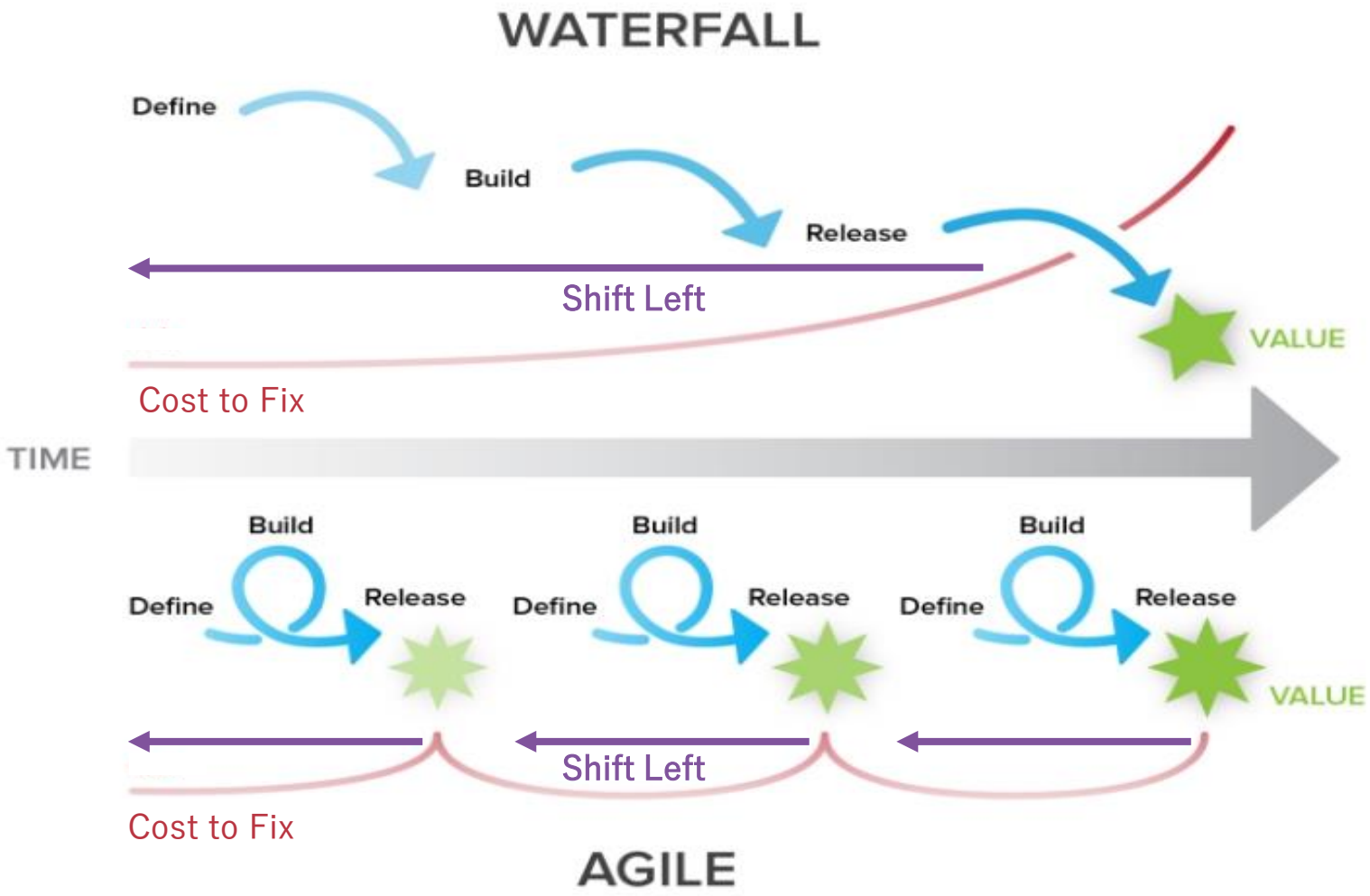
```java
public class RunGoodDBConnection2 {

    public static void main(String[] args) {

        try (
                Connection connection = DriverManager.getConnection(
                "jdbc:oracle:thin:@127.0.0.1:1521:TESTDB", "USER01","PASSWORD");
                Statement statement = connection.createStatement();
                ResultSet resultSet = statement.executeQuery("select * from config");
            )
        {
            // Do something
            statement.close();
            connection.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }

    }
}
```


参考URL：”FIO04-J. 不要になったリソースは解放する”．JPCERT CC．https://www.jpcert.or.jp/java-rules/fio04-j.html, (2019-06-01)

シフトレフト

アジャイル開発プロセス自体に、テスト工程前倒しのシフトレフトの効果がある

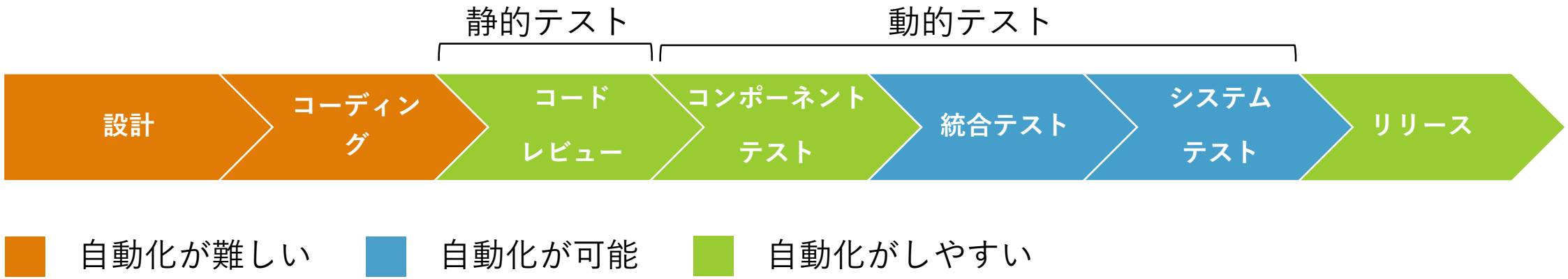
さらにイテレーション内でもシフトレフトをする



アジャイル開発プロセスでシフトレフトする

イテレーションごとに積み重なる技術的負債を低減できるよう

開発ライフサイクル内の**早期のテスト工程**から自動化する



※テスト工程の名称は、JSTQBのシラバスで定義されているテストレベル名を使用

システムテストの自動化

- 利点
  - UIテストはキャプチャリプレイ機能で基本的なフローはすぐに組める
  - 手動ではボトルネックとなる画面操作の部分を時間短縮できる
  - テスト内に繰り返し操作が多ければ多いほど自動化の効果が出る
- 注意点
  - Selenium等の特殊な経験を必要とする（キャプチャリプレイだけでは終わらない）
  - テストコードのメンテナンス性が低く、デバッグ時間が多く取られる （依存するライブラリも複雑）
  - 非機能テスト、ユーザビリティのテストは難しい（ 手動テストはなくならない[1)] ）
  - 画面キャプチャ系、クロスブラウザをやり出すとはまる

1) ”テスト自動化の8原則”．テスト自動化研究会．https://sites.google.com/site/testautomationresearch/test_automation_principle, (2019-06-01)

コンポーネントテストの自動化（ユニットテストの自動化）

- 利点
  - 実行環境準備含めて、システムテストの自動化より取り組みやすい
  - コーディングとほぼ同時に実装することで早期での設計の見直しがしやすい
  - 自動システムテストより実行時間が短い
  - ふるまいを変えないリファクタリング時に良いデバッガになる
- 注意点
  - テストコーディングが必要になる（かかる労力は本コードの設計に依存する）
  - バグ密度は多くの場合、無意味になる（テストが通るようにコーディングするため）
  - テスト品質の基準を設けるのが難しい（高いC0/C1のカバレッジは高いテスト品質を示さない）
  - セッションオブジェクト、DB接続、 DI関連のクラスなどが絡むとスタブ、モックだらけになる
  - ふるまいが変わる場合は、修正箇所（コード、テストコード）が増える

コードレビューの自動化（ソースコード静的解析）

- 利点
  - テスト工程として最も早い段階に位置する
  - コーディングが必要なく、導入が容易である
  - 負債になり得るテストコードが発生しない
  - 客観的な品質基準を設けられる
- 注意点
  - ビジネスロジックの誤りは検出できない
  - コード解析するための実行時間は加算される
  - 誤検出は発生するため、検出内容の確認が必要になる

1) ”テスト自動化の8原則”．テスト自動化研究会．https://sites.google.com/site/testautomationresearch/test_automation_principle, (2019-06-01)

各種テストの自動化に関する比較

	静的解析	コンポーネントテスト	システムテスト
自動化のしやすさ	**コマンド実行で実施可能**	テストコーディングが必要	より高度なテストコーディングが必要
実行時間	**短い**	**短い**	長い
継続的なメンテナンス負荷	**低い**	中くらい	高い
客観的な品質基準	**なりやすい**	なりにくい	なりにくい
最終的なリリース判断基準	ならない	**なる**	**なる**
False Positive（正しいコードを不具合と誤検出）	ある	**自動実行時では基本的にはない**	**自動実行時では基本的にはない**
False Negative（不具合を検出しない）	ある（ツールのチェックパターン不足）	ある（テストコーディングミス）	ある（テストケース不足、アサーション対象不足）

各種テストが検出するバグ

自動テストだけでは見つからないバグはある

	静的解析	コンポーネントテスト	システムテスト
自動テストで見つけられるバグの例	• デッドコード • コピペミス • Nullチェック漏れ • コンポーネントレベルのリソースリーク • コンポーネントレベルの非スレッドセーフ処理 ※Coverityの場合	• デッドコード • 分岐条件のミス	• 一連の操作のあとのUI表示のバグ • 実行環境に依存して発生する表示の差異
手動でないと見つけにくいバグの例	• コンポーネントレベルのビジネスロジックの記述ミス • 「美しくない」ソースコード、デザインパターンの検討	• 根本的な設計ミス • 想定していない例外が発生した場合の挙動	• 想定していない操作で発生するエラー • 非スレッドセーフ処理 • 特殊な条件（データ、システム時間）下で発生する問題

静的解析の自動化から開始する利点

- 早期に客観的な品質基準として利用できる
- 大きなバグにもつながりうる技術的負債を早期から返済できる
- テストコードの開発の必要がなく、自動化がしやすい
- 負債になりうるテストコードをメンテナンスする必要がない

静的解析でバグを見つけられるか

静的解析の技術は進歩している

	第1世代 （70～90年代）	第2世代 （90年代～2000年代初頭）	第3世代 （2003年以降）
特徴	• コンパイラより厳しい文法チェック	• パターンマッチングによるコーディングルール準拠のチェック • 限られた範囲でのパス解析	• 実行時のエラー発見に注力 • 関数間全パス解析（フロー解析） • SATソルバーを用いた高精度の解析
ノイズや誤検出	90%以上	50%以上	**15%以下**

静的解析ツールCoverity

SYNOPSYS®
COVERITY

クラス最高レベルの静的コード解析

脆弱性、クラッシュ、メンテナンスの問題になる前に、
コーディングの段階で、コードの重要な不具合と
セキュリティの弱点を検出します。

SASTのトップポジション

以下で高い評価

- 精度
- SDLCへの組み込み

FIGURE 3 Forrester Wave™: Static Application Security Testing, Q4 2017

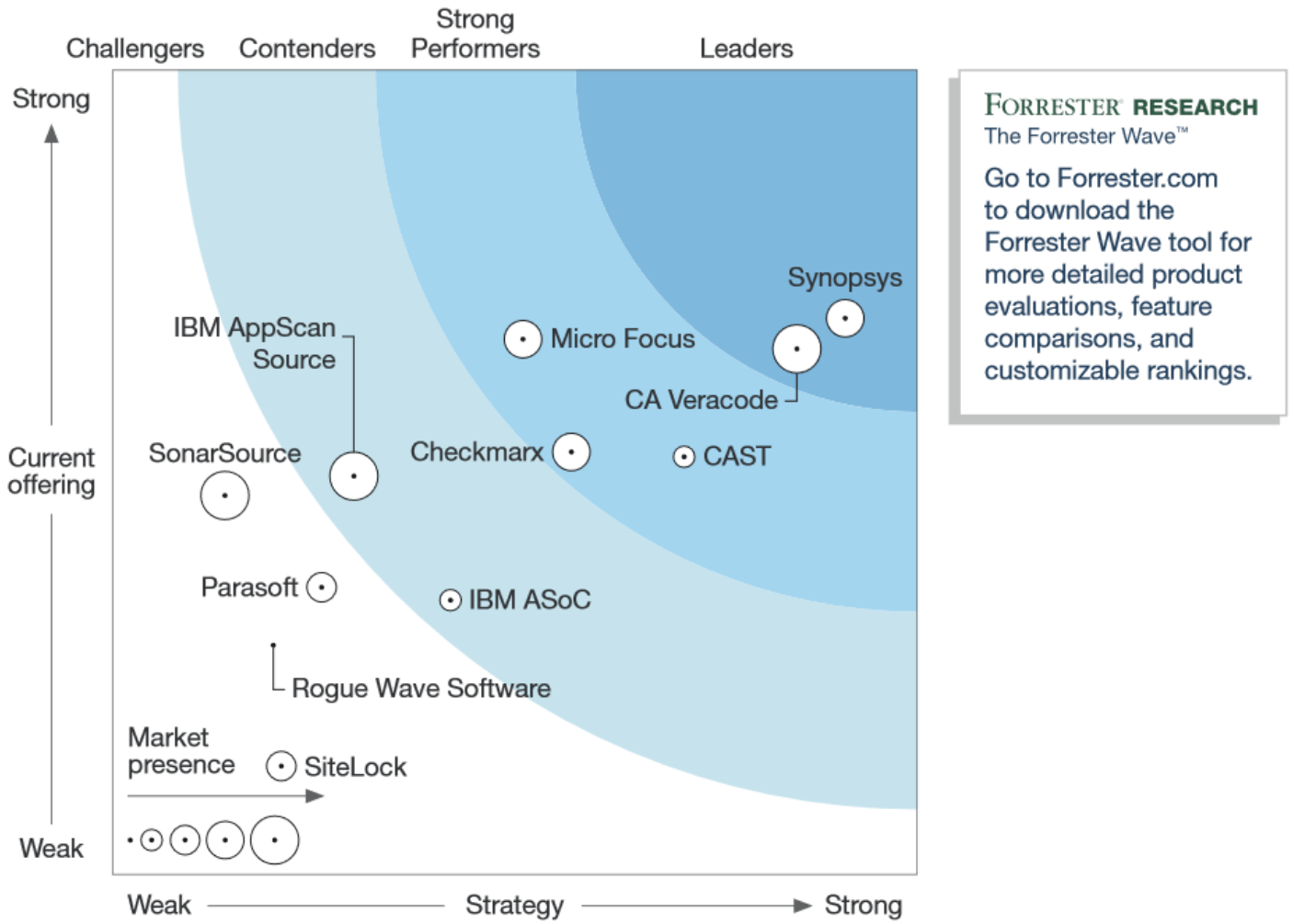


FIGURE 3. The Forrester Wave™: Static Application Security Testing, Q4 2017, December 12, 2017

Coverity サポート言語、環境

Android SDK

TypeScript

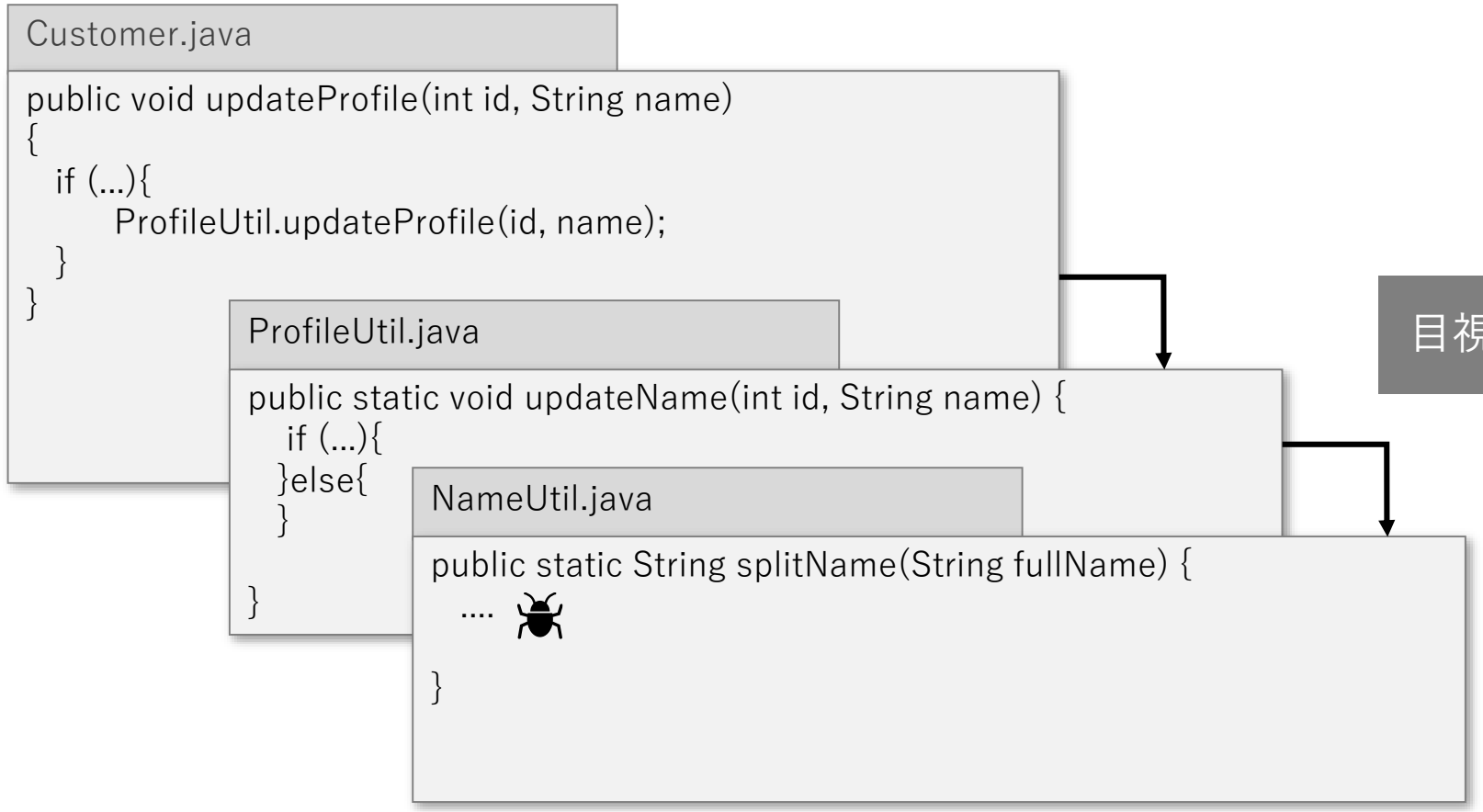
1

正確な不具合検出

Coverity

プロシージャ間の解析と全パス解析

クラス、関数、ファイルの境界を超えて、変数の状態も観察し、
全パス分岐を解析する



目視では発見困難な不具合を検出

Nullの間接参照

```
1. 分岐条件 classInstance != null は、false に分岐しました。
            if (classInstance != null)
                    return classInstance;

            Object o = null;
            try {
2. returned_null: getModuleClassLoader は null を返しています (14 箇所のうち 9 箇所で確認されています)。 [詳細を隠す]
CID 18186 (#1/1): null 戻り値の間接参照 (NULL_RETURNS)
3. null_method_call: null オブジェクト org.openmrs.module.ModuleFactory.getModuleClassLoader(getModule()) のメソッドが呼ばれています。
                Class<?> c = ModuleFactory.getModuleClassLoader(getModule()).loadClass(getClassName());
/var/lib/jenkins/jobs/historical_openmrs/workspace/openmrs/api/src/main/java/org/openmrs/module/ModuleFactory.java
1172        }
1173
1174        /**
1175         * Get a module's classloader
1176         *
1177         * @param mod Module to fetch the class loader for
1178         * @return ModuleClassLoader pertaining to this module. Returns null if the module is not
1179         *         started
1180         * @throws ModuleException if the module does not have a registered classloader
1181         */
1182        public static ModuleClassLoader getModuleClassLoader(Module mod) throws ModuleException {
1183                ModuleClassLoader mcl = getModuleClassLoaderMap().get(mod);
1184
    1. 分岐条件 mcl == null は、true に分岐しました。
    2. var_tested_null: mcl は null です。
1185                if (mcl == null)
1186                        log.debug("Module classloader not found for module with id: " + mod.getModuleId());
1187
    3. return_null_var: nullである mcl を返しています。
1188                return mcl;
1189        }
```

統計的な解析により、コードベース全体で、このメソッドのNullチェックがどれくらい行われているかを表示

呼び出し先のメソッドの中身を表示

Nullを返却していることが分かる

スレッドのデッドロック

1. **lock_acquire:** ロック ManagerEvents.class を獲得しています。

```
public static synchronized void modifyPrice(PosScreen pos) {
    PosTransaction trans = PosTransaction.getCurrentTx(pos.getSession());
    String sku = null;
    try {
```

CID 14261 (#1/1): スレッドのデッドロック (LOCK_INVERSION)

2. **lock_order:** ManagerEvents.class を保持した状態でのロック MenuEvents.class の獲得は、他の場所でのロック順序と競合しています。 [詳細の表示]

```
        sku = MenuEvents.getSelectedItem(pos);
    } catch (ArrayIndexOutOfBoundsException e) {
    }
```

ロック取得の順序が間違っています

1 **lock_acquire**	ManagerEvents.java:73
2 **lock_order**	ManagerEvents.java:77
2.1 **getlock**	MenuEvents.java:423

ロック取得が正しい順序で行われている例

A1 **lock_acquire**	MenuEvents.java:86
A2 **example_lock_order**	MenuEvents.java:104
A2.1 **getlock**	ManagerEvents.java:268
B1 **lock_acquire**	MenuEvents.java:86
B2 **example_lock_order**	MenuEvents.java:102
B2.1 **getlock**	ManagerEvents.java:423

他の場所ではどのような順序でロックを取得しているかを記載し、ロックが発生する原因を突き止める

Coverityで検出可能な主な不具合と脆弱性

クラッシュの原因

- Null ポインタの間接参照
- ポインタの解放後のメモリ使用
- 二重解放
- 配列の新規作成と削除の不一致

プログラムの不正な動作

- デッドコード
- 未初期化変数
- 負の変数の無効な使用
- コピー＆ペーストのミス

並列処理の問題

- デッドロック
- 競合状態
- ブロック呼び出しの誤用
- メモリ破損

パフォーマンスの低下

- メモリ・リーク
- スタックの使用度合い

セキュリティ上の脆弱性

- バッファ・オーバーフロー
- API エラー処理
- XSS / インジェクション / CSRFなどの Web アプリケーション脆弱性
- Mobile IDの誤用などのAndroidセキュリティ
- Java Script用Google Cloud Storage APIsとAWS SDKのセキュリティ

1

正確な不具合検出

誤検出 < 15% を実現

Coverity

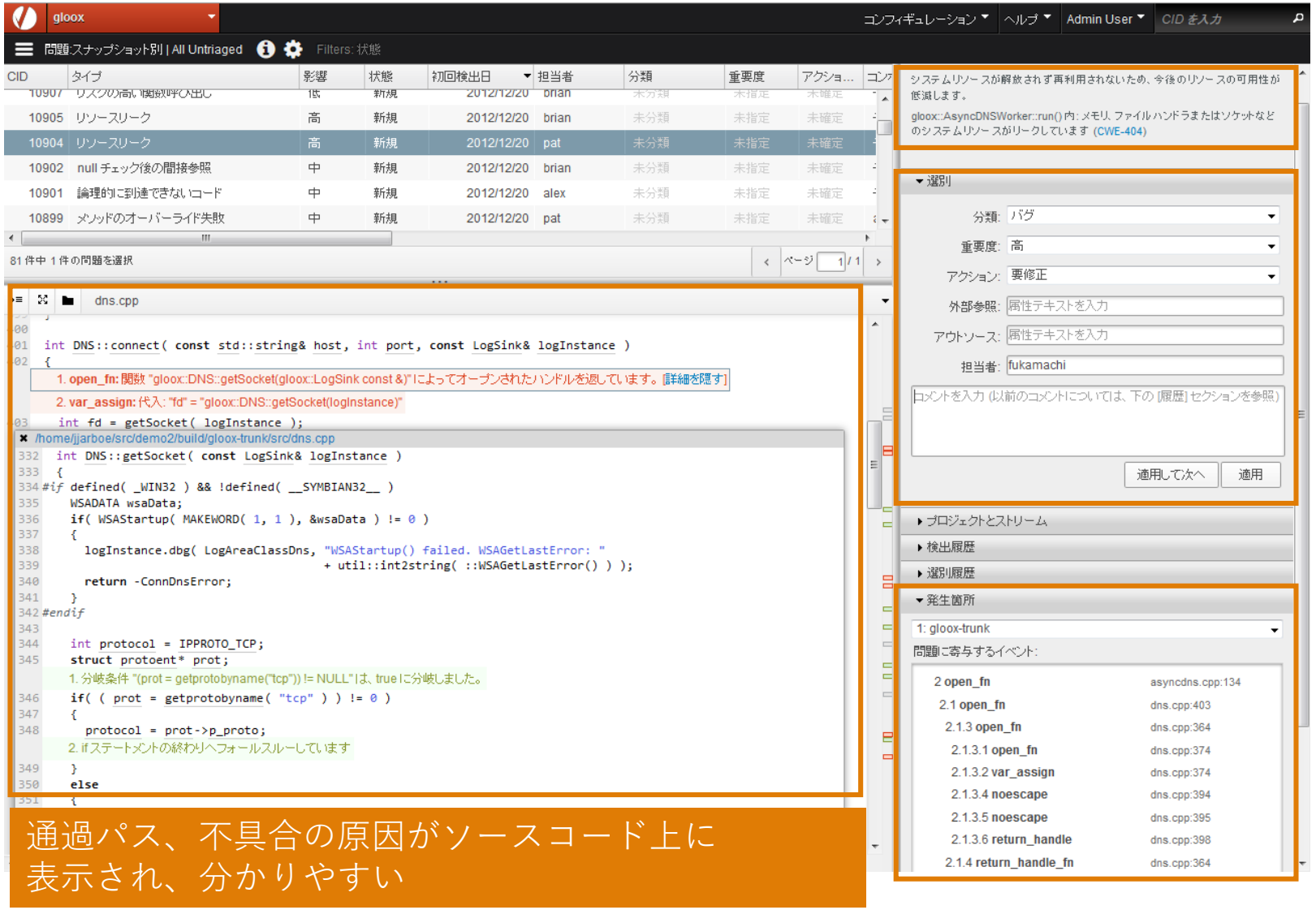
Coverity

1 正確な不具合検出

誤検出 < 15% を実現

2 素早い問題の把握と修正

レビューを効率化し、不具合管理を容易にするレポート機能



通過パス、不具合の原因がソースコード上に表示され、分かりやすい

CWE 情報

不具合管理のためのワークフロー情報

不具合発生までの主要なイベントや、類似の問題の発生個所へのリンク

Coverity

1 正確な不具合検出

誤検出 < 15% を実現

2 素早い問題の把握と修正

開発者が使い続けられる静的解析

Coverityユーザ様の事例紹介

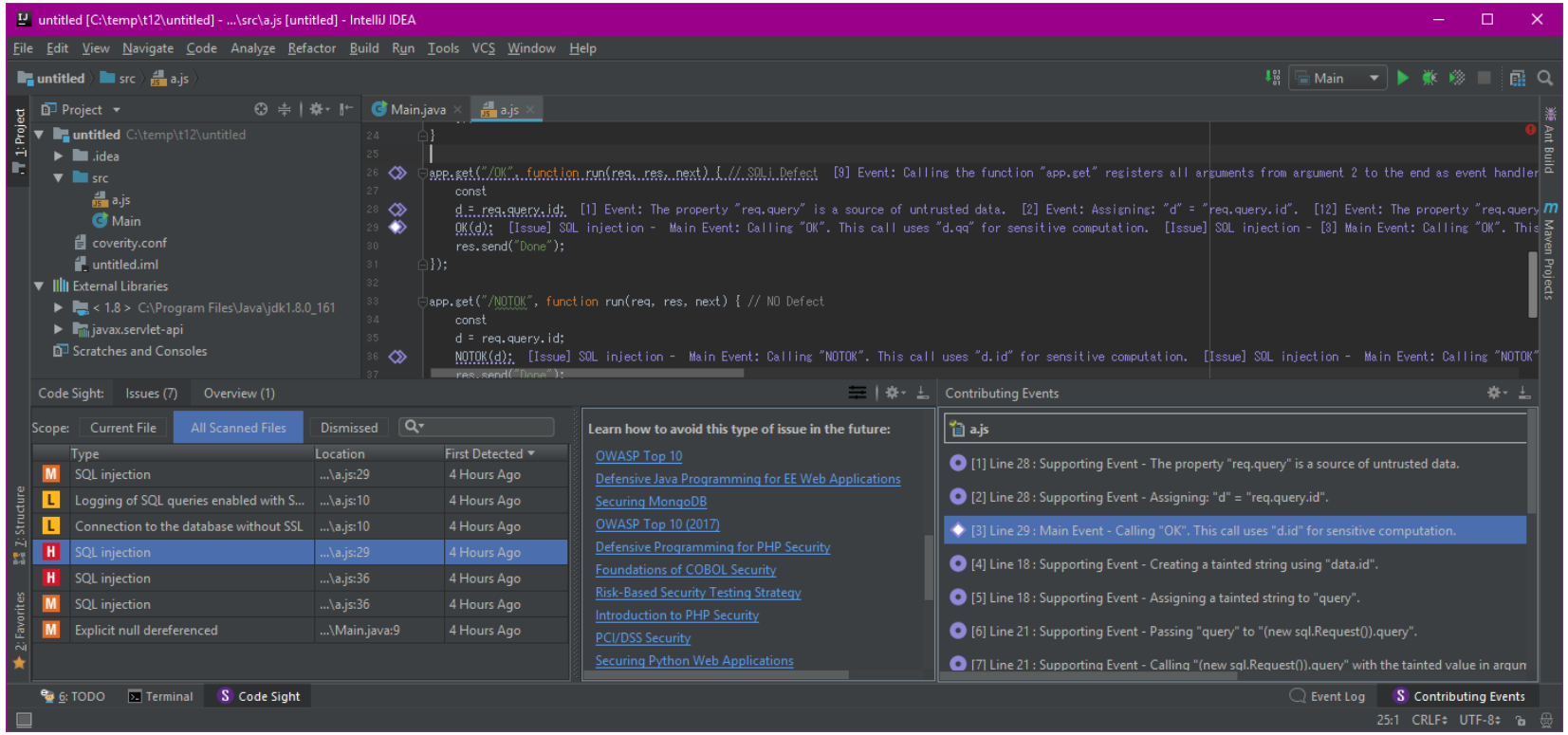
セガ	『静的解析ツールがバグを潰し、新人を育てる』
オムロンソフトウェア	『高精度なエラー検出能力はスマホ向け開発現場でも大好評』
NTTデータイントラマート	『大規模業務アプリケーション開発の効率化とソースコードの標準化、品質向上に貢献』
DeNA	『開発効率を向上させ、魅力あるゲームづくりに注力DeNAが実現した マンパワーに依存しないソフト品質管理』
三菱電機	『静的解析ツール活用は顧客満足のため』
GMOクリック証券	『コードに潜む不具合の可能性を確実に検知金融サービスシステムの安定稼働を実現』
SmartFocus	『SmartFocus、シノプシスを採用して市場投入までの期間を短縮
日本光電	『日本光電が取り組むソフトウェア品質向上、ヒューマンエラーを未然に防ぐ仕組みとは？』
アルカテル・ルーセント	『アルカテル・ルーセント』

事例入手URL：https://www.synopsys.com/ja-jp/software-integrity/resources/case-studies.html

静的解析導入のポイント

IDEでも静的解析する

コーディングしているそばから**リアルタイム**に問題を解消するには、
IDEの静的解析プラグインを導入する



静的解析のノイズを軽減し、根付かせる

開発者が**使い続けられる**ことが重要

- 開発初期から使用し、日々の解析自動化をする
  - 開発者の1人の開発量はせいぜい1k行/日、不具合密度を多くて2.0件/k行とすると、
    1人の1日当たり指摘1～2件程度の検出

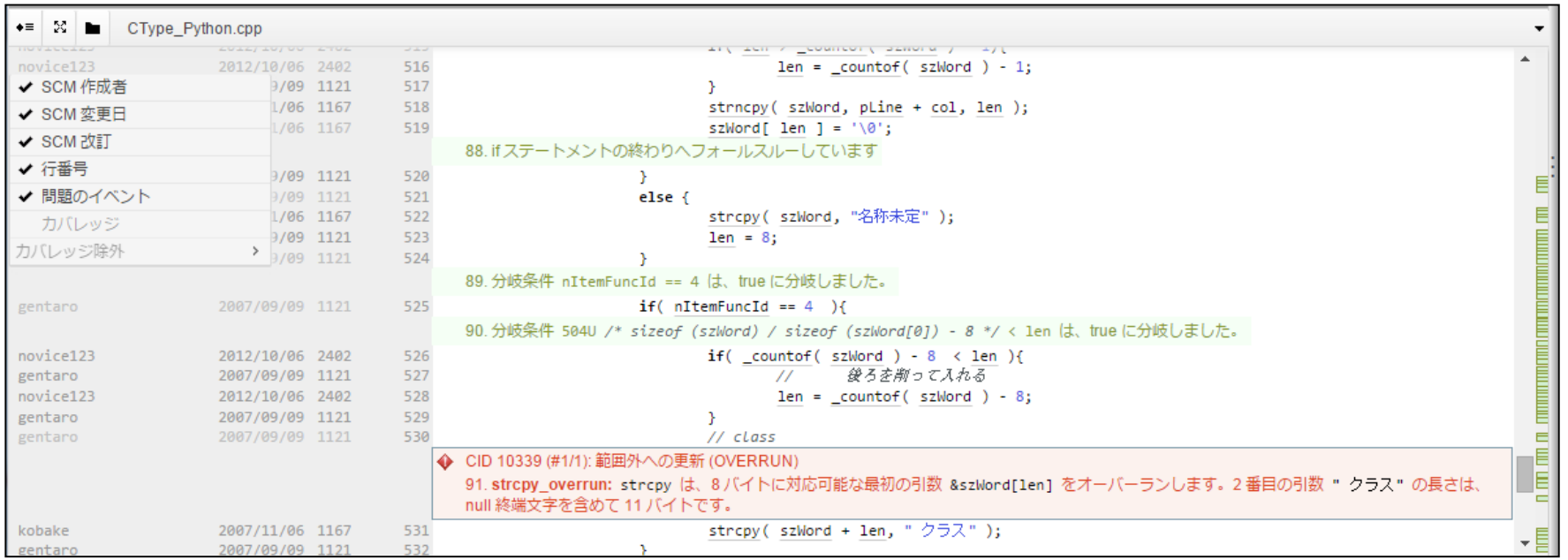
- 重要な部分のみを修正する運用ルールとする
  - Coverity：　影響度高と中のみ修正、コーディング規約チェックオプションを追加しない
  - SonarQube： BLOCKERとCRITICALレベルの指摘のみ修正、 Code Smellsは無視

- 誤検出の少ない静的解析ツールCoverityを使用する

不具合の自動アサイン

不具合検出→修正→再解析のサイクルを定常化させるには、

SCMのユーザ情報から**自動的に修正担当者をアサイン**する仕組みにする



まとめ

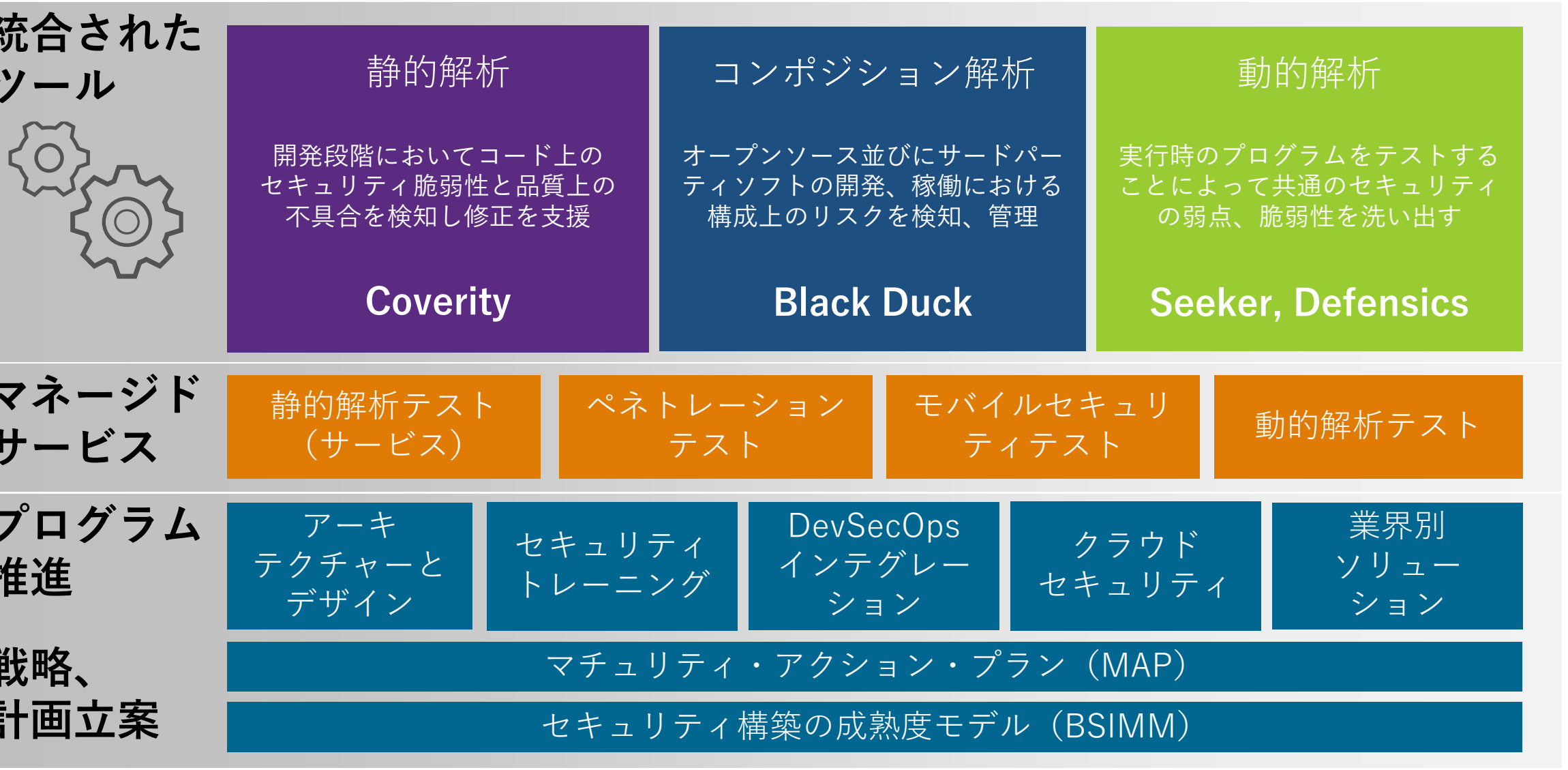
早期から品質担保するには、ソースコード静的解析が重要

- 自動化可能なプロセスで一番レフトのタイミングにある
- 導入が簡単なため、早期の品質の底上げ、可視化ができる
- 静的解析でないと検出できないバグが存在する

Coverityなら

- 正確な不具合検出で、誤検出 < 15% を実現
- 素早い問題の把握と修正で、開発者が使い続けられる静的解析

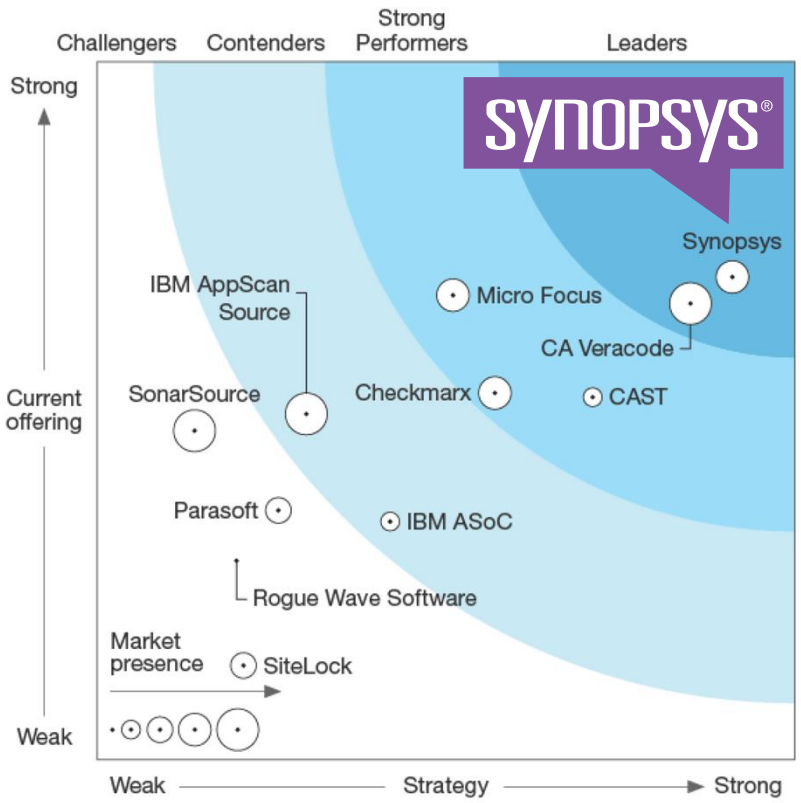
シノプシスが提供するソリューション



市場でのポジション（静的解析＆コンポジション解析）

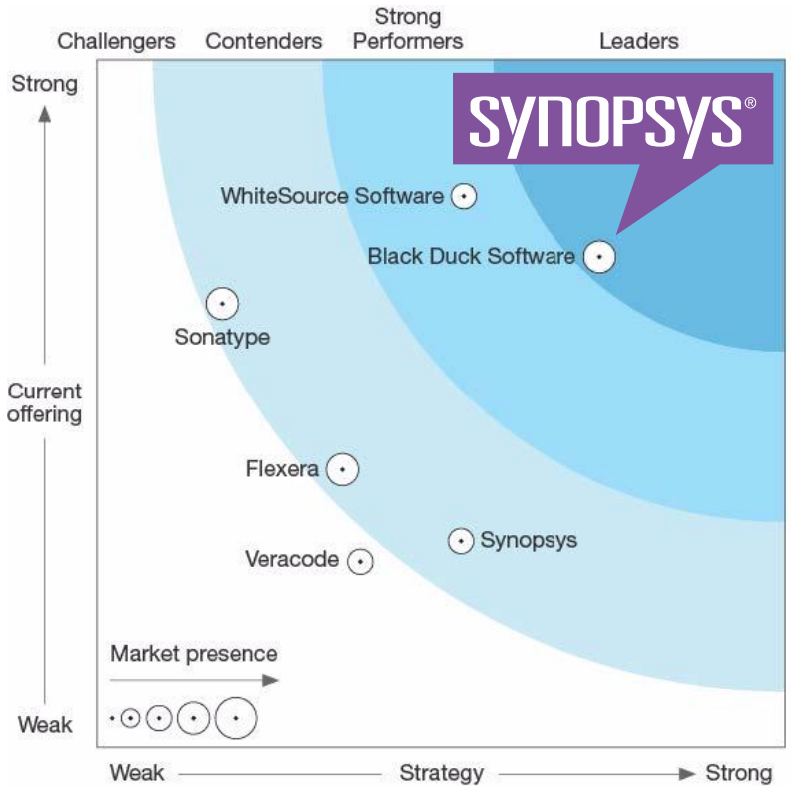
シノプシスは静的解析分野とコンポジション解析分野の両分野でのトップポジション

Forrester Wave
Static Application Security Testing



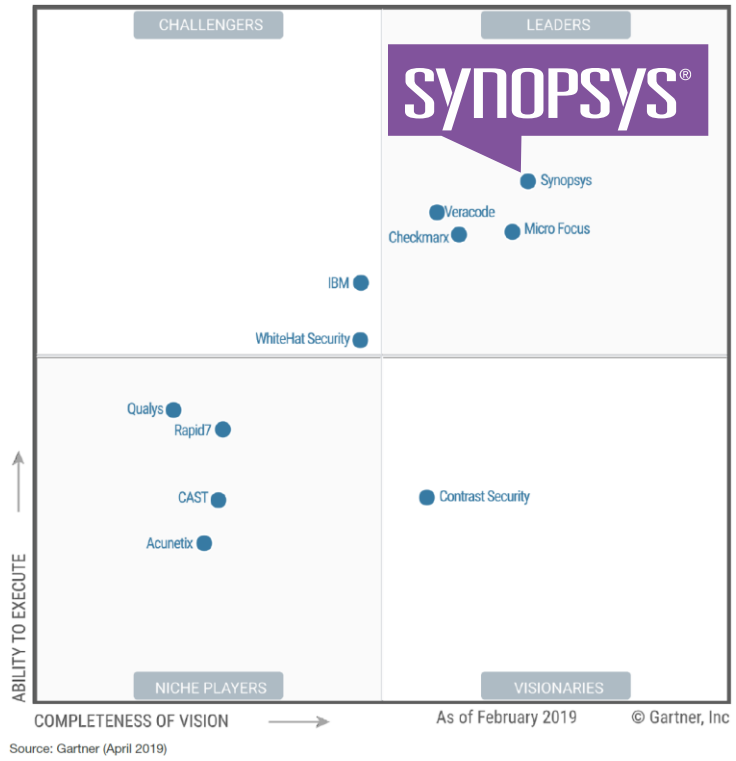
Q4 2017, December 12, 2017

Forrester Wave
Software Composition Analysis



Q1 2017, February 23, 2017

Gartner Magic Quadrant
Application Security Testing



April, 2019

世界のトップ企業4,000社＋がお客様です

トップ20行中16行
銀行

10社中9社
独立系ソフトウェアベンダ

10企業/団体中9企業/団体
航空・宇宙・防衛企業

10社中8社
グローバル・ブランド

10社中6社
半導体企業