テスト自動化を促進する周辺技術とその運用法

コベリティジャパン株式会社

coverity®

動的解析と静的解析

- 動的解析
  - 利点
    - 見つかった問題はすべて修正対象（誤検知はない）
    - パフォーマンスや発生しうるタイミングの解析が可能
  - 欠点：
    - 実際に動作させる必要がある＝解析自体に人的リソースが必要＝自動化は困難
    - 問題が見つかってもその原因の特定が困難な場合がある

- 静的解析
  - 利点
    - 全自動、実行環境は必要ない
    - 人的には困難な全制御パスのシミュレーションが可能
    - 人間が介在しないため客観的な品質データを取得可能
  - 欠点
    - 誤検知が発生する＝ソースコードが存在しないライブラリなど
    - パフォーマンス、タイミングは解析できない

coverity®

静的解析の種類

- コーディング規約チェック系
  - ベスト・プラクティスに基づいたコードの書き方
  - 標準規約など予め定義したコーディングルールに対し違反がないかを確認

- ランタイムエラー検出系
  - バグ自体を検出
  - 従来、実行時でないと確認できなかった、メモリリーク、データ破壊、データ競合などを静的に検出

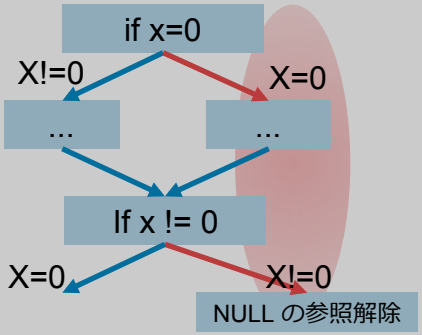
- コベリティの強さ
  - スピード（差分ビルド、差分解析）
  - 精度、根本原因の特定のし易さ
  - 検出した問題の管理（派生開発サポート、フローに基づくバグ管理）

coverity®

静的解析 – 手戻りを激減させるテクノロジー

SAT ソルバ

不具合が発生しないパスを排除し、誤検知を減らす



全パス & プロシージャ間解析

関数またはファイルに跨る不具合の検出

```
void bar() {
foo(p);
if(p != 0) { ... }
}
```

```
void foo(int *p) {
*p = 42;
}
```

統計的な解析

プログラムの目的を推測し、予期しない動作を報告する

```
int *p = malloc(sizeof(int));
if(p != 0)  *p = 42;
...
int *p = malloc(sizeof(int));
if(p != 0)  *p = 42;
```

多分正しい

```
int *p =
        malloc(sizeof(int));
*p = 42;
```

多分間違い

ホワイトボックス ファザー

汚染データを投入し、無害化されているかを自動的に確認する

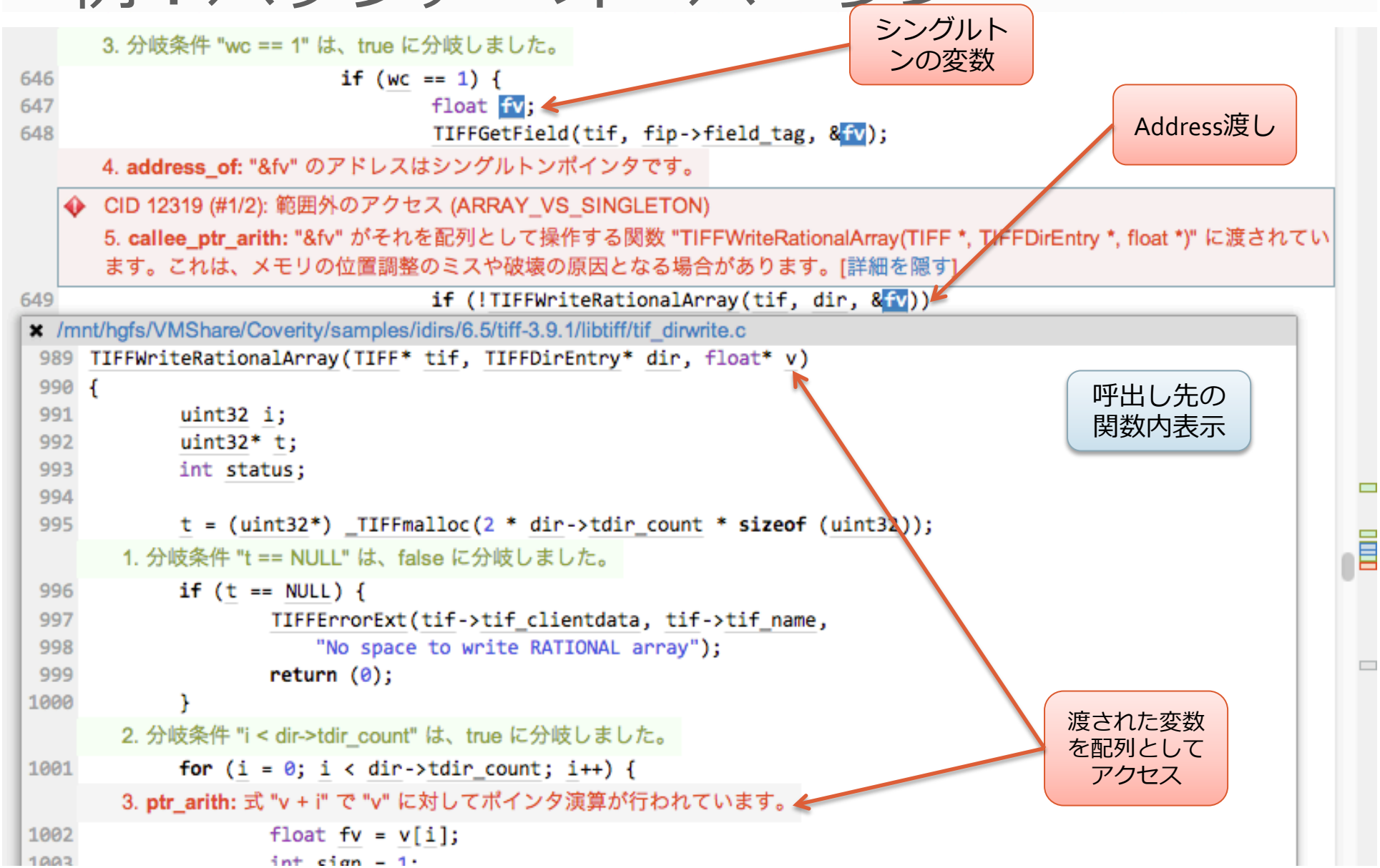
a b c d ... < > & ' ...

htmlEncode()

< > &
" '

coverity®

例：バッファーオーバーラン



coverity

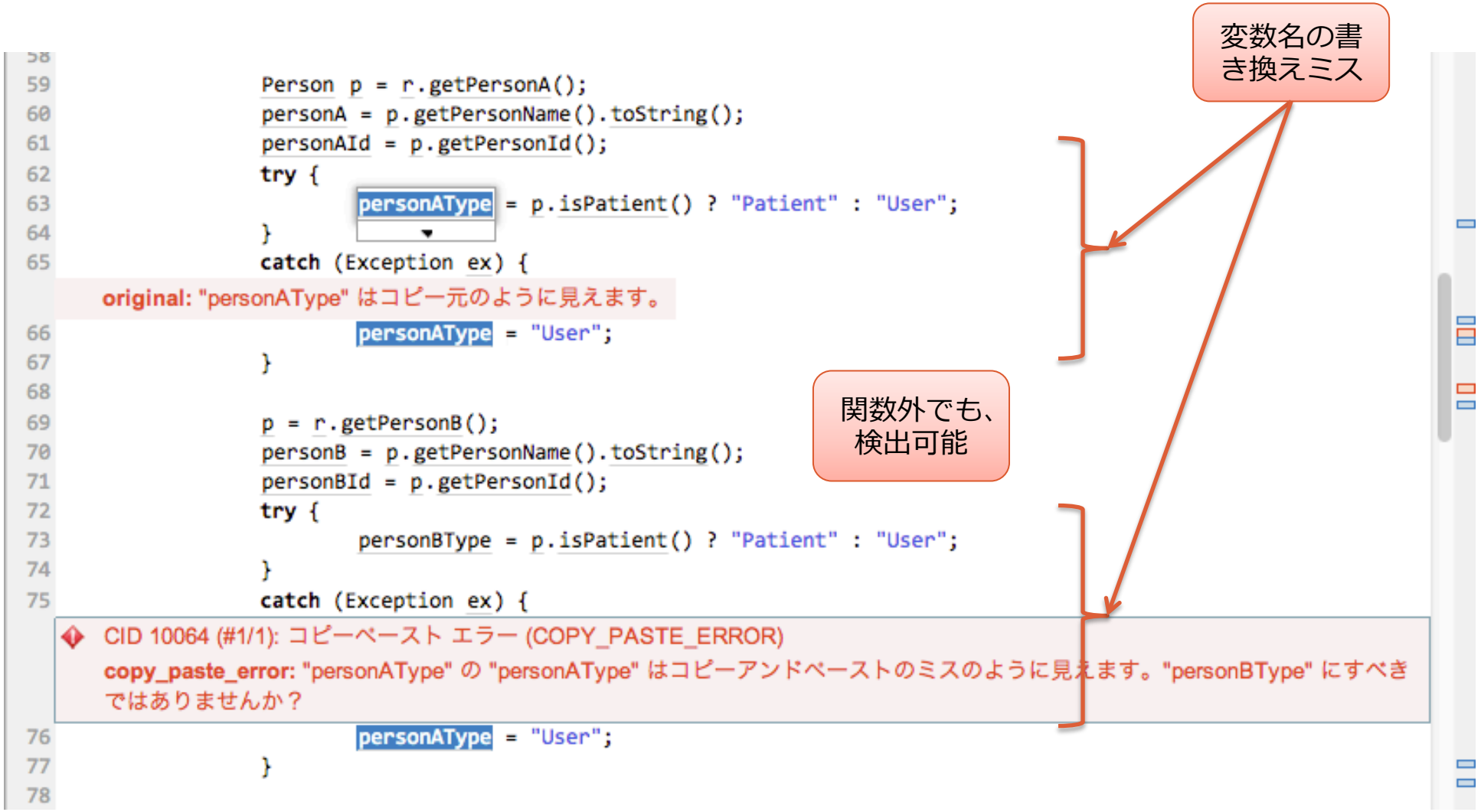
例：リソースリーク

呼出し先の関数内でアロケートされた領域を検知

```
5257 static int
5258 TIFFFetchStripThing(TIFF* tif, TIFFDirEntry* dir, uint32 nstrips, uint64** lpp)
5259 {
5260         static const char module[] = "TIFFFetchStripThing";
5261         enum TIFFReadDirEntryErr err;
5262         uint64* data;
     1. alloc_arg: "TIFFReadDirEntryLong8Array(TIFF *, TIFFDirEntry *, uint64 **)" は、 "data" に格納したメモリをアロケートしています。[詳細の表示]
5263         err=TIFFReadDirEntryLong8Array(tif,dir,&data);
     2. 分岐条件 "err != TIFFReadDirEntryErrOk" は、false に分岐しました。
5264         if (err!=TIFFReadDirEntryErrOk)
5265         {
5266                 TIFFReadDirEntryOutputErr(tif,err,module,TIFFFieldWithTag(tif,dir->tdir_tag)->field_name,0);
5267                 return(0);
5268         }
     3. 分岐条件 "dir->tdir_count != (uint64)nstrips" は、true に分岐しました。
5269         if (dir->tdir_count!=(uint64)nstrips)
5270         {
5271                 uint64* resizeddata;
5272                 resizeddata=(uint64*)_TIFFCheckMalloc(tif,nstrips,sizeof(uint64),"for strip array");
     4. 分岐条件 "resizeddata == NULL" は、true に分岐しました。
5273                 if (resizeddata==0)
     CID 12334 (#1/1): リソースリーク (RESOURCE_LEAK)
     5. leaked_storage: 変数 "data" がスコープを外れるため、リソースがリークしています。
5274                         return(0);
5275                 if (dir->tdir_count<(uint64)nstrips)
5276                 {
5277                         _TIFFmemcpy(resizeddata,data,(uint32)dir->tdir_count*sizeof(uint64));
5278                         _TIFFmemset(resizeddata+(uint32)dir->tdir_count,0,(nstrips-(uint32)dir->tdir_count)*sizeof(uint64));
5279                 }
5280                 else
5281                         _TIFFmemcpy(resizeddata,data,nstrips*sizeof(uint64));
5282                 _TIFFfree(data);
5283                 data=resizeddata;
```

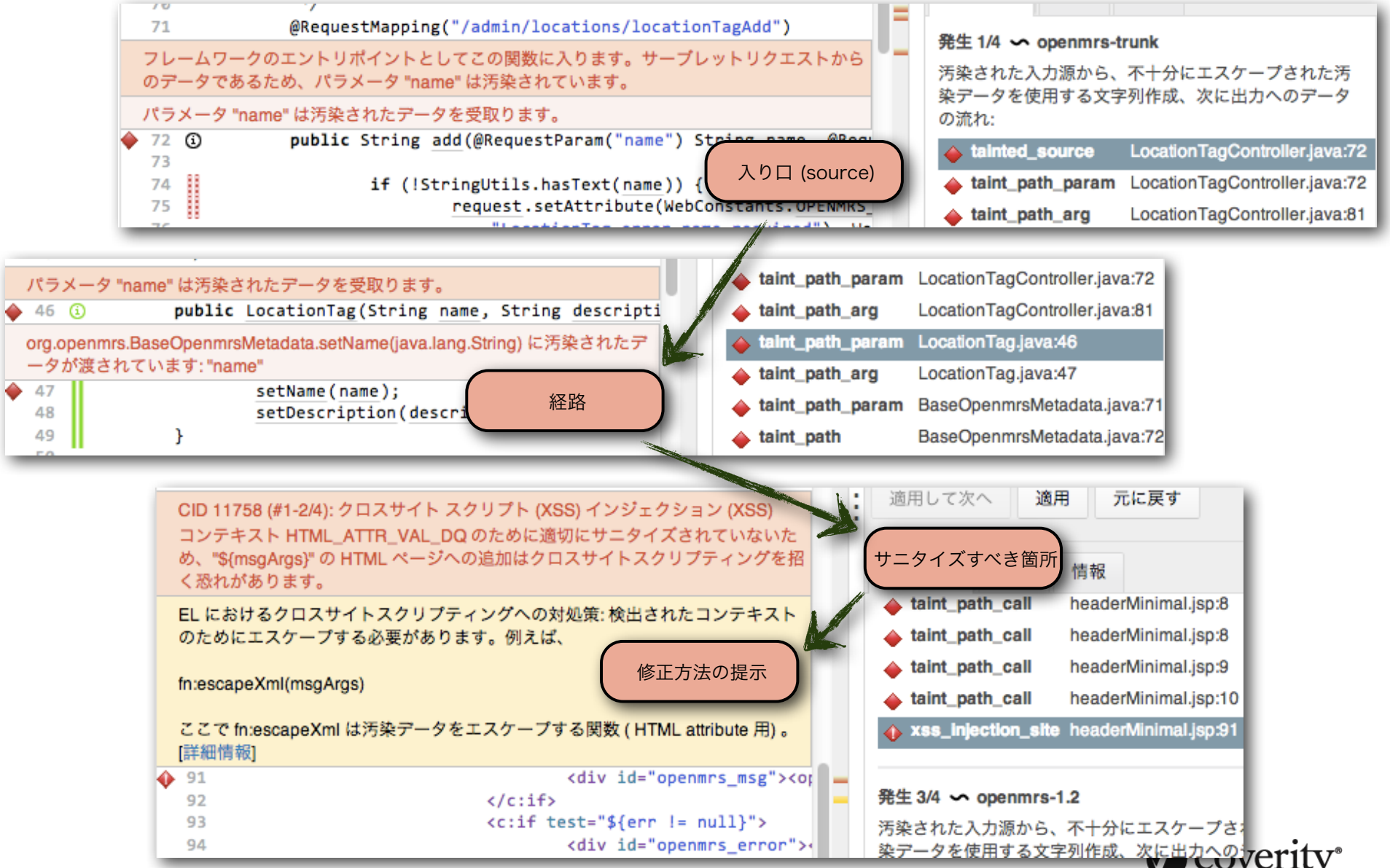
領域をを解放せずにローカル変数を破棄

例：コピペミス



coverity

例：クロスサイトスクリプティング



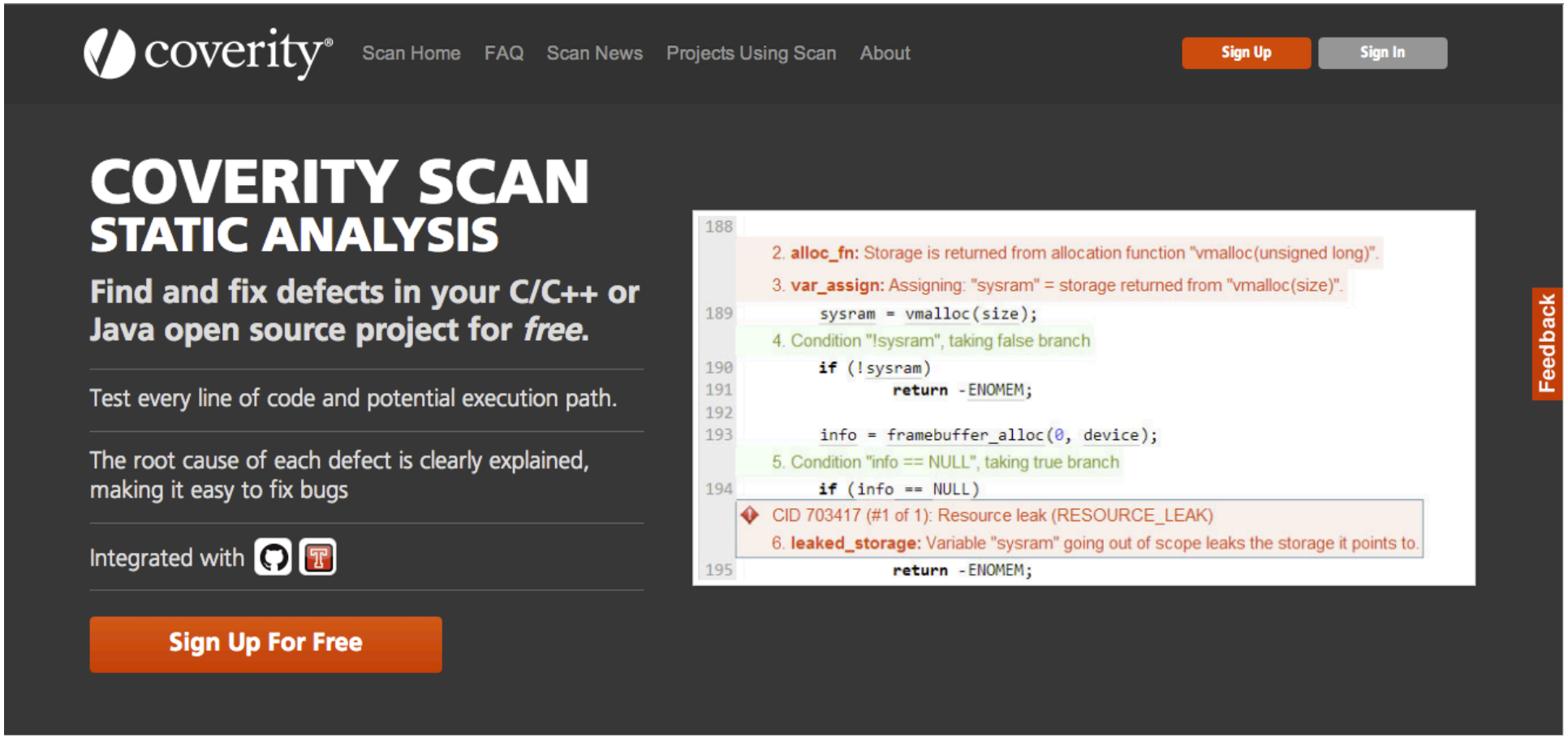
検出した問題をトラッキングする

開発組織、プロジェクト、サプライヤ

Coverity を取り入れた開発者ワークフロー

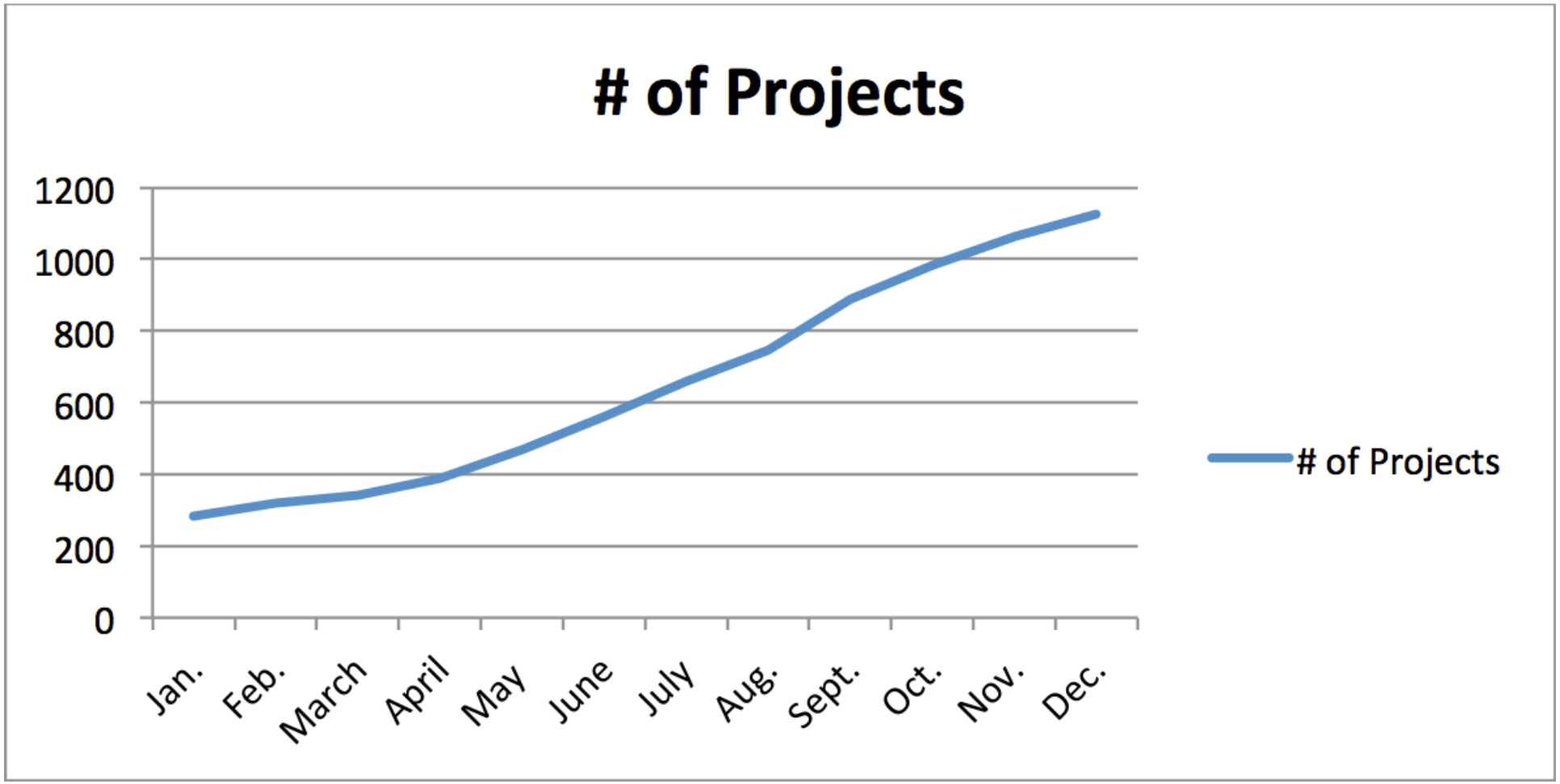
coverity®

SCAN: オープン・ソースの状況



coverity

２０１３年に利用者が急増



coverity®

オープン・ソースの規模感

Table 1: 2013 OSS コードの規模感: C/C++			
サイズ (行数)	2013 プロジェクト数	2013 プロジェクトの割合	2012 プロジェクトの割合
100,000 以下	358	48%	38%
100,000-499,999	299	40%	44%
500,000-100 万行	42	6%	7%
100 万行以上	42	6%	11%
計	**741**		

coverity®

検出されたバグの密度平均（年別）

Table 2: 2008-2013 平均不具合検出率	
Coverity Scan Report 年	平均不具合密度 C/C++ （件/KLOC）
2008	.30
2009	.25
2010	.81
2011	.45
2012	.69
2013	.59

Coverity によって自動検出されたバグのうち、影響度が高・中レベルのもの

coverity®

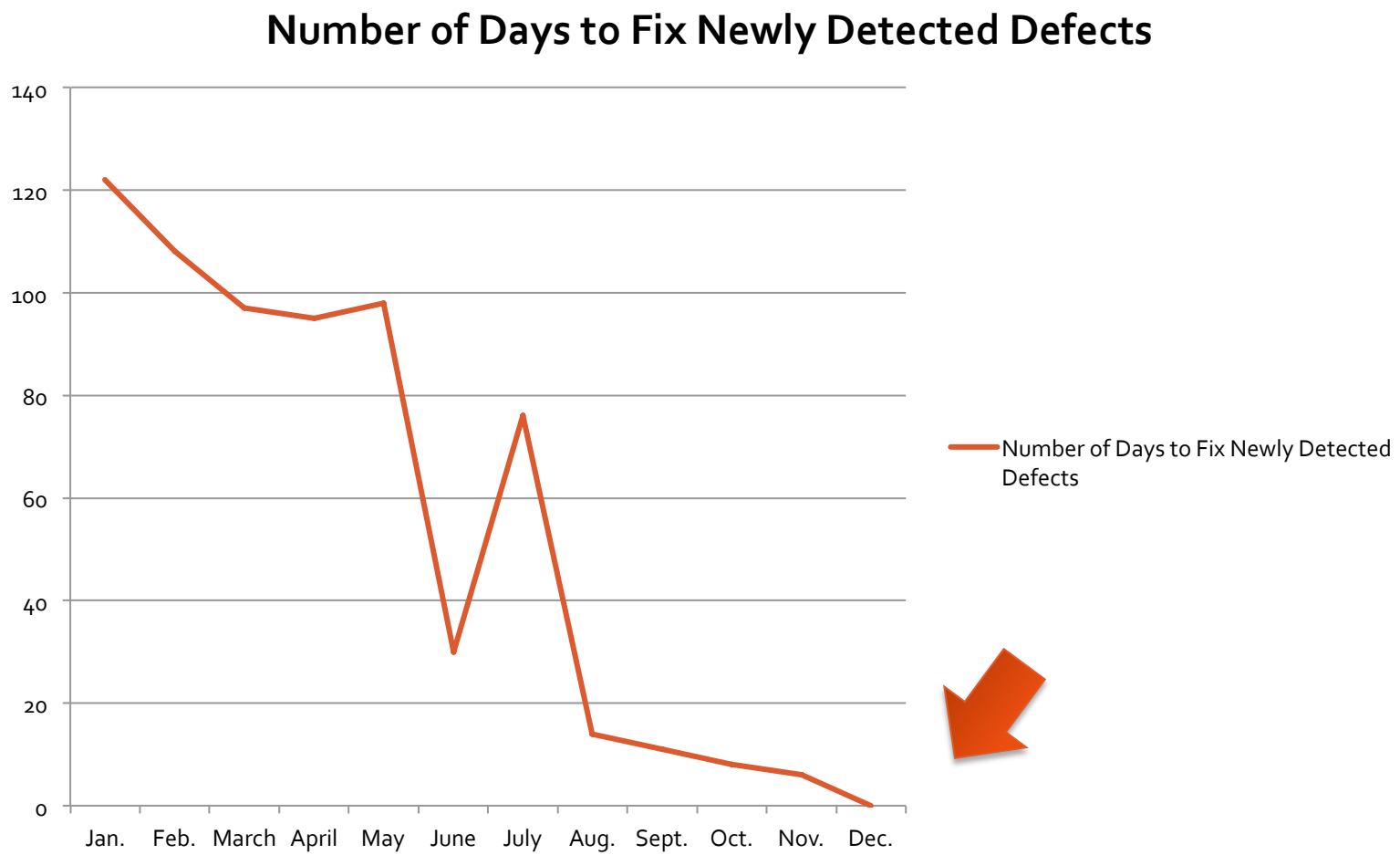
Linux の現在

Table 8: LINUX ANALYSIS: 2006-2013				
年	**バージョン**	**解析行数**	**新規不具合数**	**修正数**
2006	2.6.16	3,451,730	1,264	435
2007	2.6.16	3,458,369	425	217
2008	2.6.27	4,202,209	596	365
2009	2.6.32	4,862,567	527	417
2010	2.6.33, 2.6.34, 2.6.35	5,504,780	3,858	462
2011	2.6.38, 2.6.39, 3.0, 3.1	6,849,378	2,331	1,283
2012	3.2, 3.3, 3.4, 3.5, 3.6, 3.7	7,387,908	5,803	5,170
2013	3.12	8,578,254	3,299	3,346

Linuxコンポーネント別不具合密度

TABLE 9: 2013 LINUX 不具合密度（コンポーネント別）	
コンポーネント	Linux 2013
Drivers	0.58
Drivers-Media	0.47
Drivers-SCSI	0.67
Staging-lustre	0.85
Drivers-Staging	0.77
Networking	0.50
Drivers-USB	0.49
Kernel	0.61
Drivers-Infiniband	0.65
Drivers-USB	0.49
Kernel	0.61
Drivers-Infiniband	0.65

Linux：不具合修正の姿勢が変わった！



2013 年から Dave Jones が Linux の Coverity SCAN 担当に

coverity®

今日は Coverity 開発チームの内情を...

coverity®

Transforming Testing Through Automation

Coverity's Journey to Automate Testing as Part of Product Development

Andreas Kuehlmann
Coverity, Inc.
San Francisco, CA, USA
akuehlmann@coverity.com

coverity®

テスト自動化
5 つのチェックポイント

coverity®

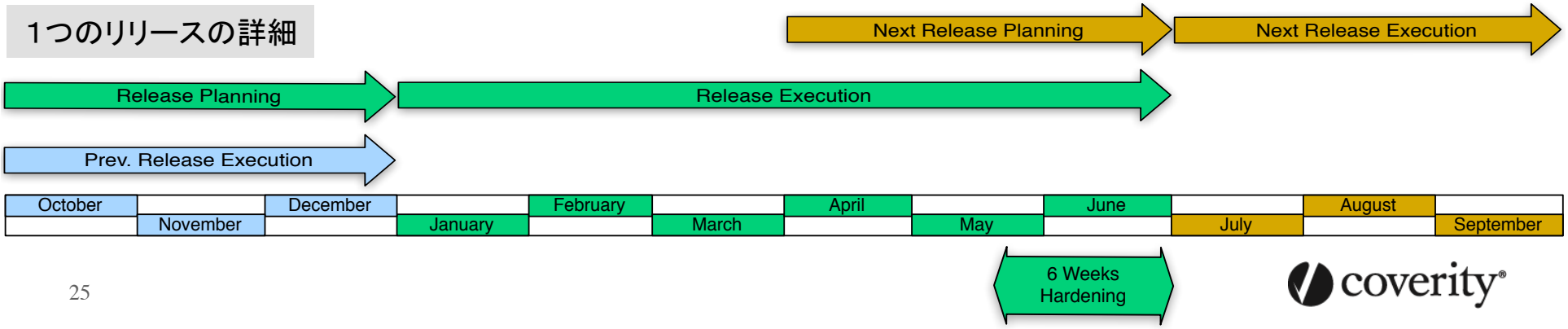
コベリティの製品リリースプロセス

一般的なリリースプロセス:

- 6 ヶ月のリリースサイクル
- 3 ヶ月の次期リリース計画
- リリース直前の 6 週間の“ハードニング” サイクル
  - 2 週間の “フィーチャーフリーズ”
  - 2 週間の “インテグレーションフリーズ”
  - 2 週間の “リリースフリーズ” と GA のためのアップロード
- リリースのコードネームはカリフォルニア州の都市名
  - Alameda, Berkeley, Davis, Eureka, Fresno,

リリースを実行するプロセス:

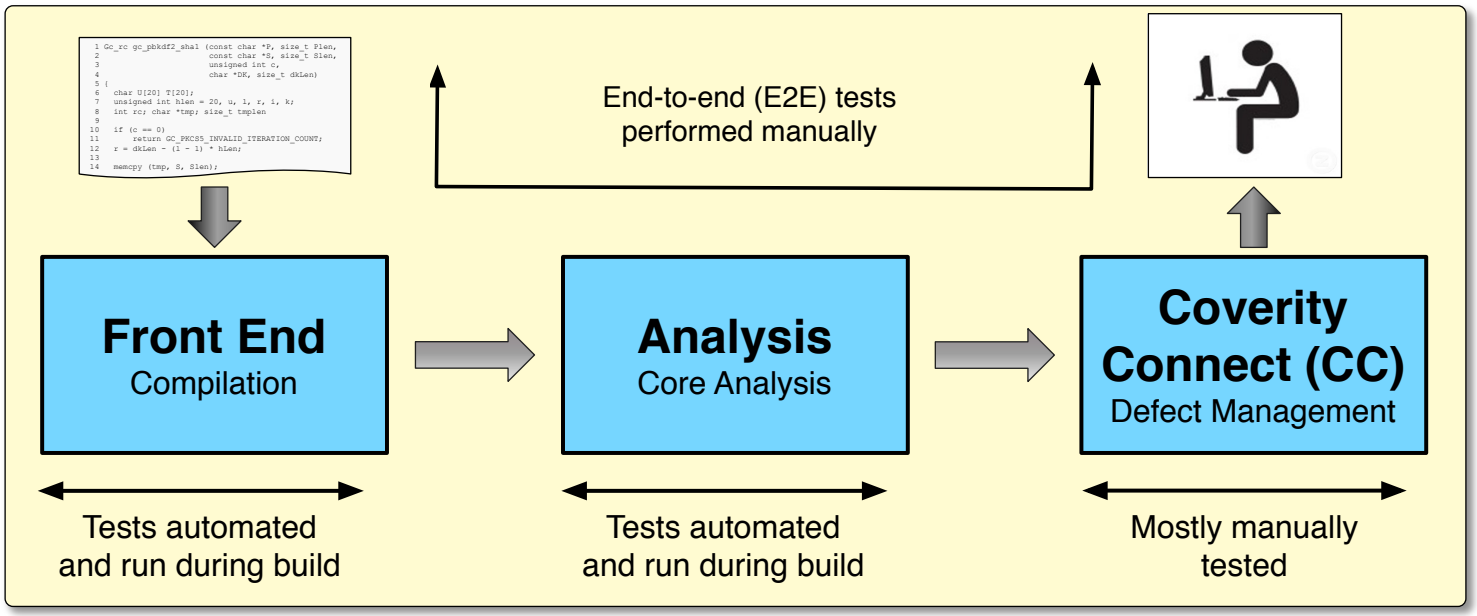
- アジャイルプロセス
- 部門横断的なスクラムチーム
- 2 週間のスプリント
- メールまたは対面での日次スタンド・アップ
- Jira と PivotalTracker を介した進捗管理
- Bugzilla によるバグ管理
- Jenkins-CI による継続的統合
- 週 1 回のステータスミーティング
  - リリースハードニングでは、週に 3 回



coverity®

2 年前の状況

コベリティの製品構造概要:



2年前のテストアプローチ:

- フロントエンドと解析のコンポーネントは自動化されたテストでテストされていた
- Coverity Connect はほとんどが手動でテスト
- End-to-End のテストは手動で行われていた

coverity

2 年前の状況

- QA プロセスは東ヨーロッパで
  アウトソースし実施していた
  - 伝統的な QA スキル、開発スキルは
    わずかかもしくは皆無
  - 10時間の時差
  - 言語的な壁

- 6 週間のハードニングサイクルは “ロシアン・ルーレット”のようなものだった
  - “フィーチャー実装完” と “フィーチャーテスト” の時間的解離
    - 開発者は品質に対して、即座の直接的な責任を感じていなかった
    - 何に出くわすかわからない – “簡単なバグ” なのか “致命的バグ” なのか
  - とるに足らないバグに QA サイクルの多くが費やされていた
  - 手動のプロセスでは、テストを 2-3 回しか回せない

- 製品品質による苦しみ
  - よい品質でリリースするために、チームは多くのストレスを抱えていた

coverity®

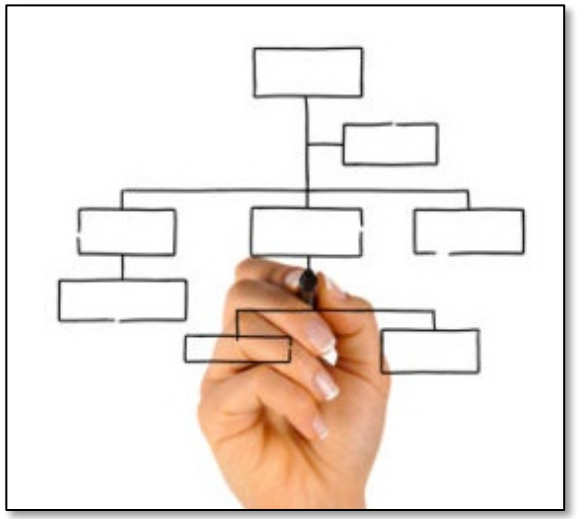
No.5：本当に自動化が効くのか？

- ６ヶ月のメジャーリリースサイクル
- サポートプラットフォームが非常に多い
  - プラットフォームの違いのみのテストケースが多い
  - Jenkins 上のビルドは各プラットフォームに分散

coverity®

矯正への最初の試み

- QAと開発の間の境界を取り除く
  - 組織的 + 時間的 (開発とテストを織り込んでいく)



- 2011 初頭に組織を再編
  - 2/3 の QA リソースを開発組織に移管
  - 目的を、テストの自動化を名目に、
    “開発しながらテストする” ように設定

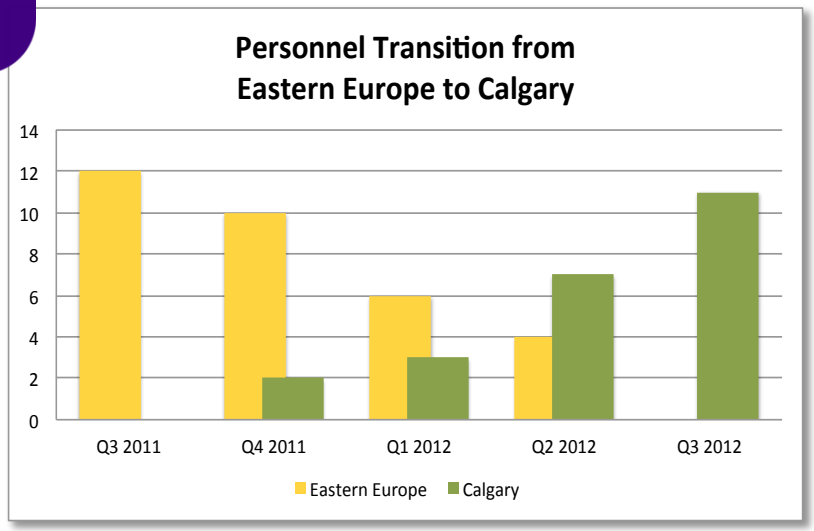
- しかし、結果は期待はずれだった
  - QA チームはプログラミングに関してトレーニングされておらず、自動テストを開発するのが困難であった
    - テストの自動化に関しては、実際ごくわずかしか進歩しなかった
  - 時差と言語の壁が存続していた

No.4:スキルセットが正しいか？

- リソースを東ヨーロッパから
  カナダ、カルガリーに移動
  - 1 時間の時差
  - サンフランシスコから 2.5 時間のフライト
  - 言語の壁がない

- 2011 年の夏に移行を開始
  - 移行は12ヶ月以上かかった
  - 多くは予算に影響を与えなかった

- スキルセットの変更
  - テスト自動化に焦点をあてたプログラミング
    スキル
  - テストの自動化に対する高いモチベーション



coverity®

モチベーションの変化

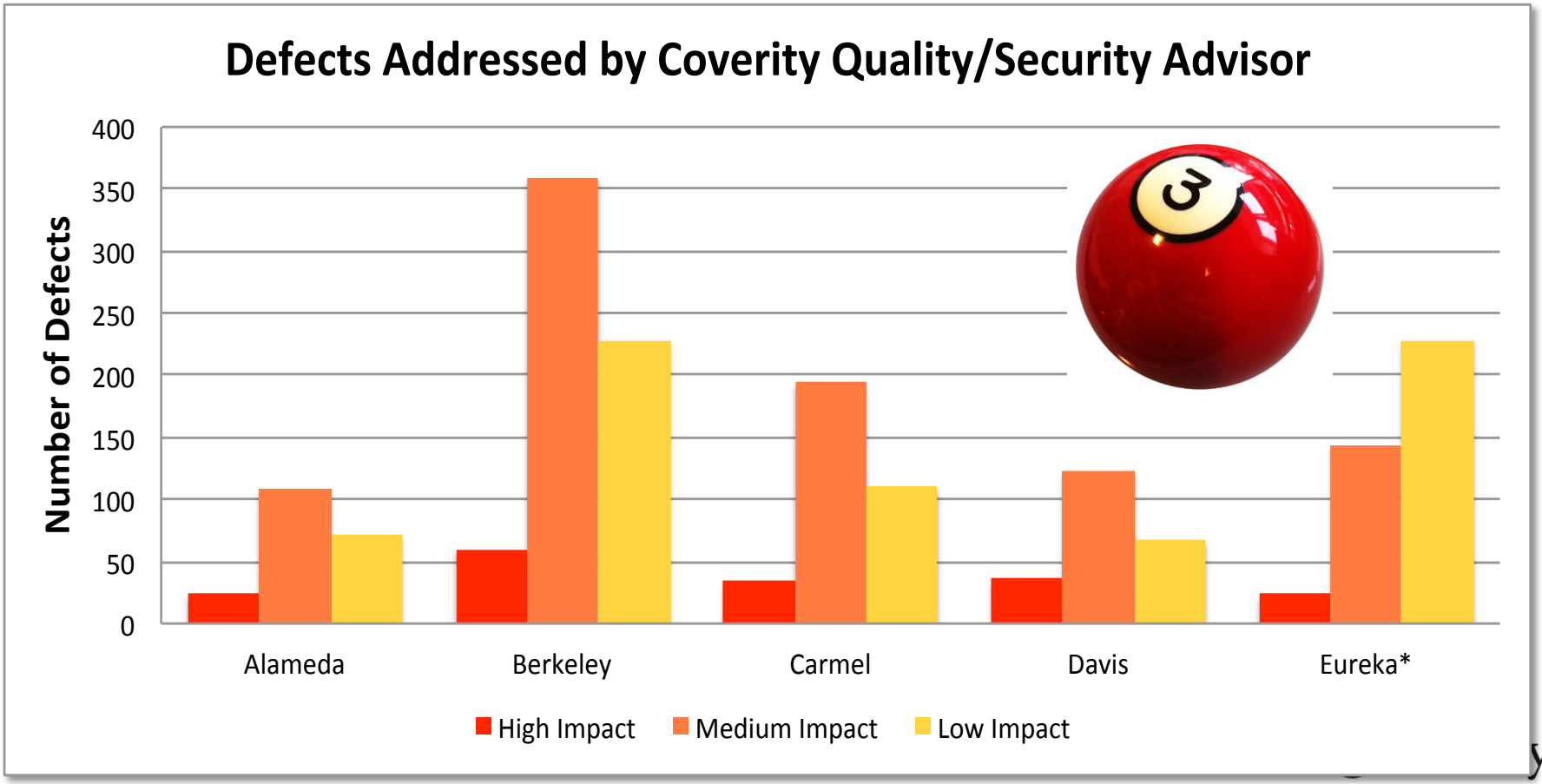
- テストの自動化に集中する。
  - とりわけ、Coverity Connect (CC) と End-to-End (E2E)

- 自動化されたテストにより２つの効果:
  - フィーチャーに期待する挙動を成文化:
    - テストと実装の間の “ネゴシエーション”
    - テストがパスすれば、フィーチャーは “完了” と宣言できる
  - 機能が完成したあとのリグレッションの検知:
    - 次のフィーチャーセットの作業をしている間の意図しない機能の変更
    - 将来のフィーチャー開発のための “ガードレール”

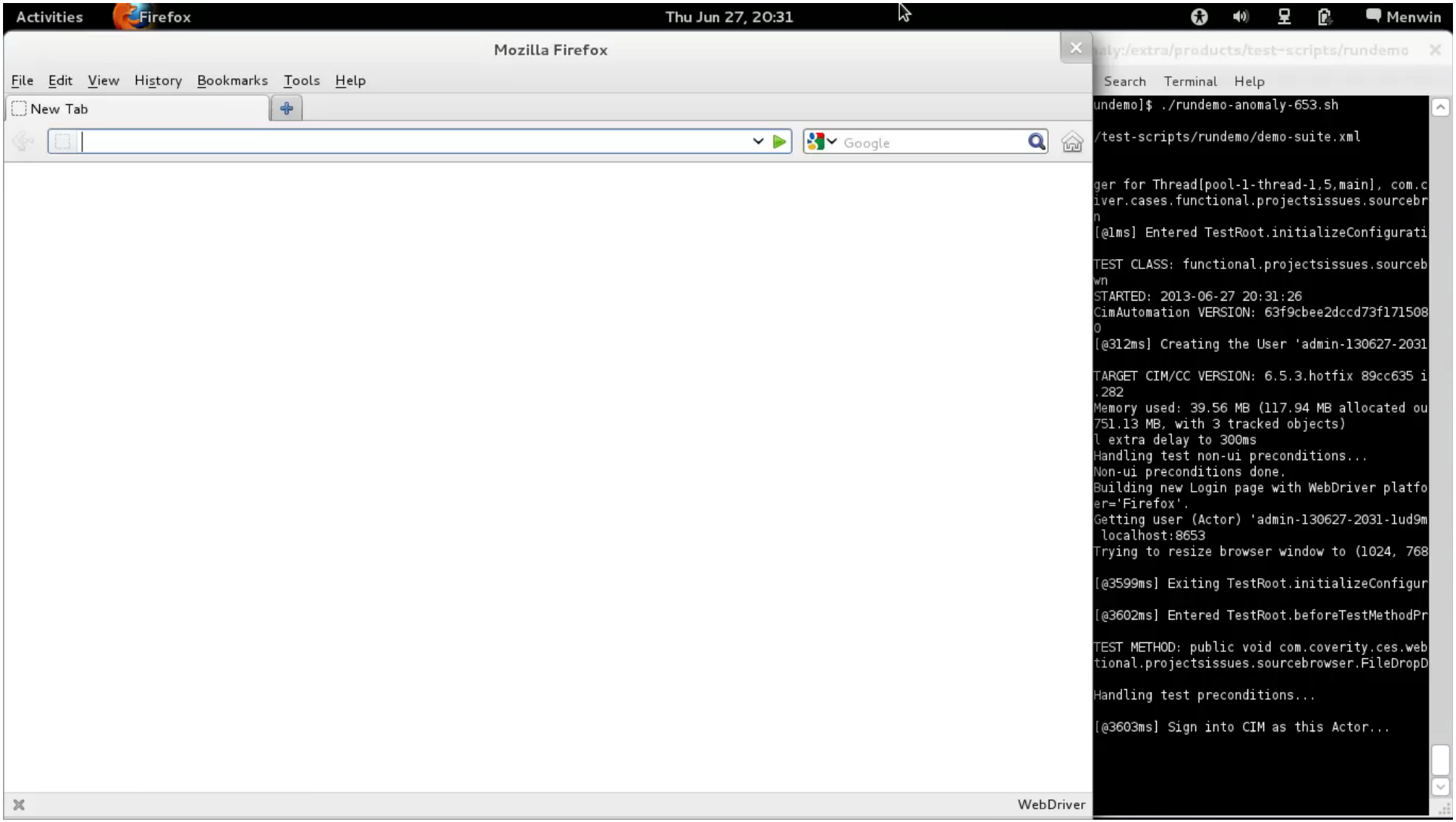
- ただし、全てのテストがコードに密に連携して開発できるわけではない
  - テストのためのよいアーキテクチャが必要となる

No.3: クリーンなコードか？

- もちろん、自社製品を開発にも自社製品を使用
- クリーンなコードでなければ、自動化も進まない



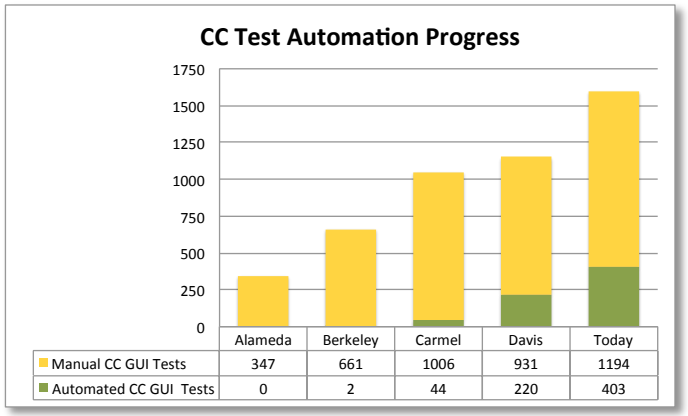
Coverity Connect の自動テストの例

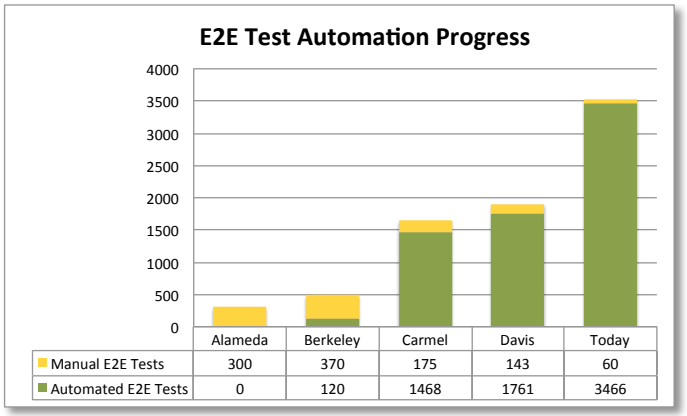


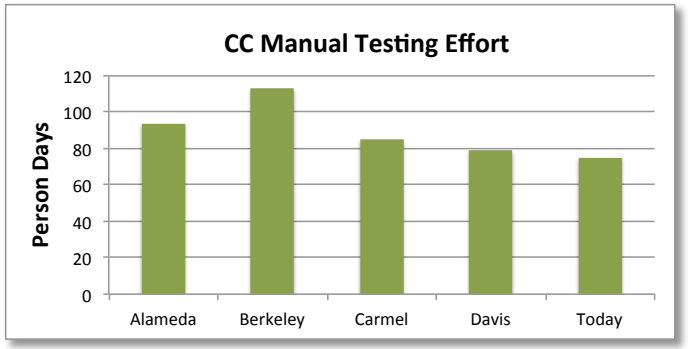
coverity®

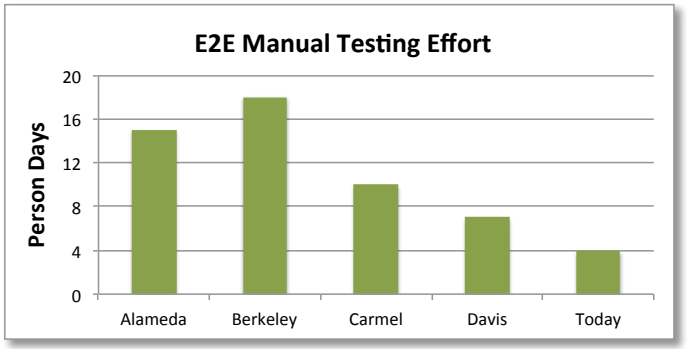
テスト自動化の進捗

- 2011年秋～2013年春までの Coverity Connect と End-to-End のテスト自動化の結果







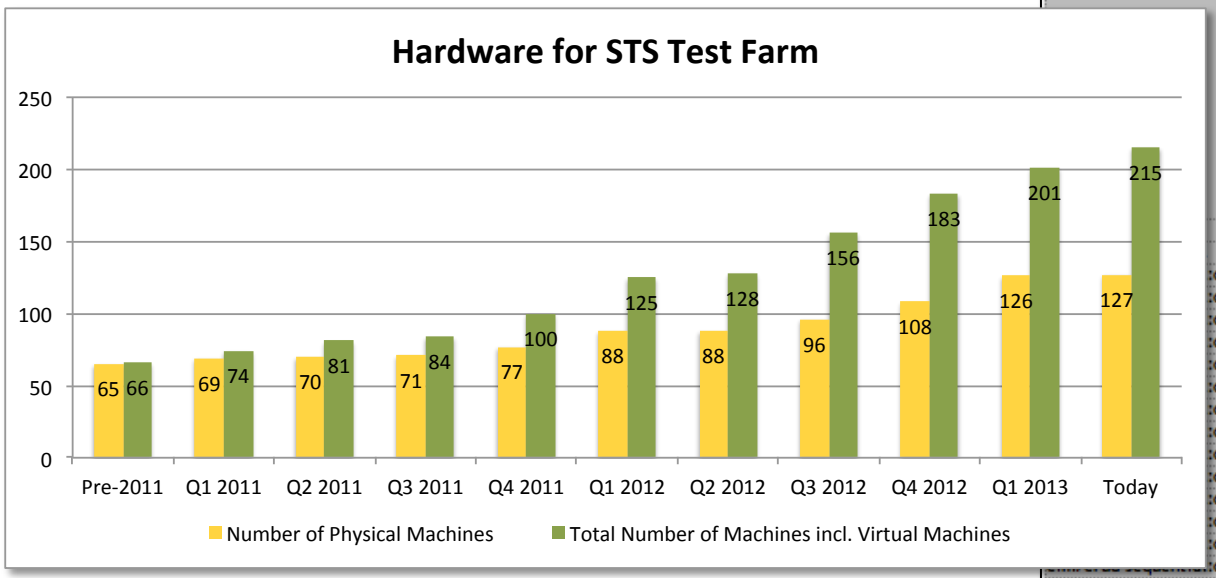


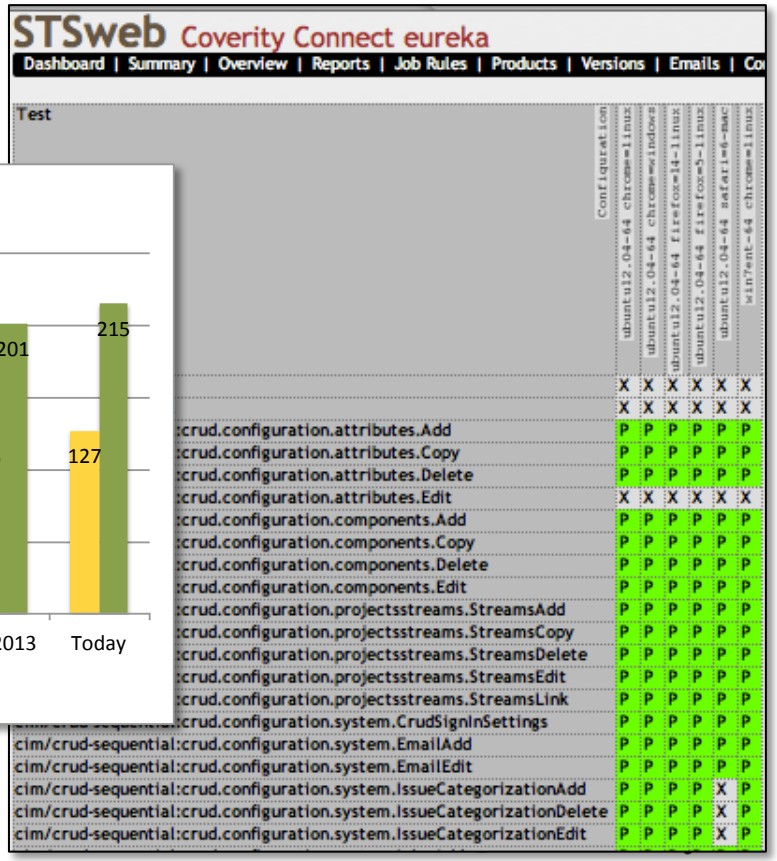
coverity®

インフラ

- 短いテストのほとんどは、Jenkins 上のビルドプロセスの一部として実行される
- 自動化テストのスケジュールとレポートのため、システムを拡張
  - すべてのプラットフォームのためのテストを

    自動化するため、サーバに大きな投資
  - システムの自動割当





自動化を支えた新しい考え方と技術...

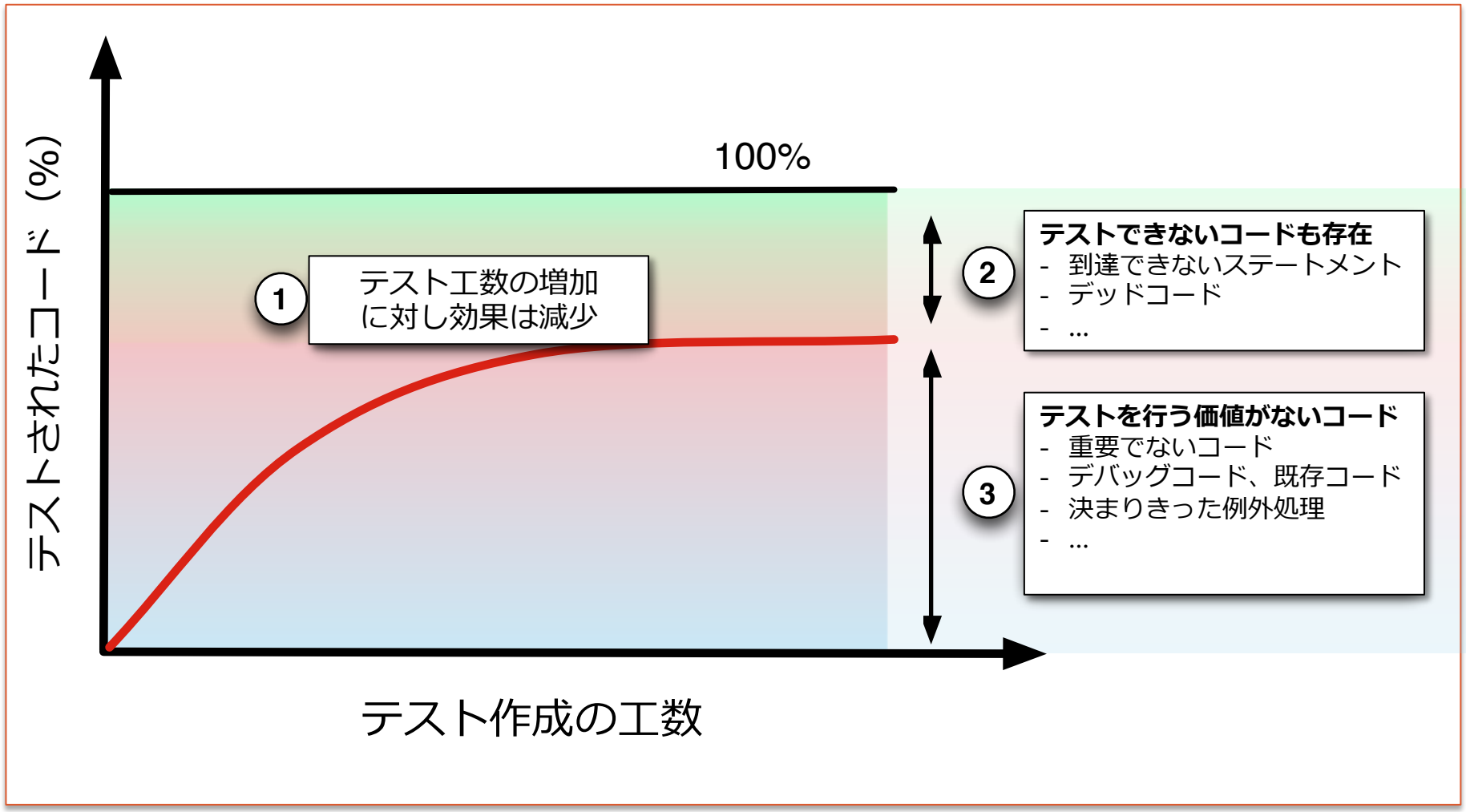
The Ultimate Machine – Claude Elwood Shannon
http://www.kugelbahn.ch/sesam_e.htm

coverity®

No. 2 : カバレッジの罠にはまらない

- 進捗を計測する良い指標がないことに気づいた
  - コードカバレッジはテスト工数に関しては良い指標ではなかった
  - 新機能のコードは、全体のカバレッジで把握しにくい
  - “開発しながらテストする”ことを牽引する計測可能な基準の欠如

コードカバレッジに対するチャレンジ



coverity®

No.1: 優先度を設定しているか？

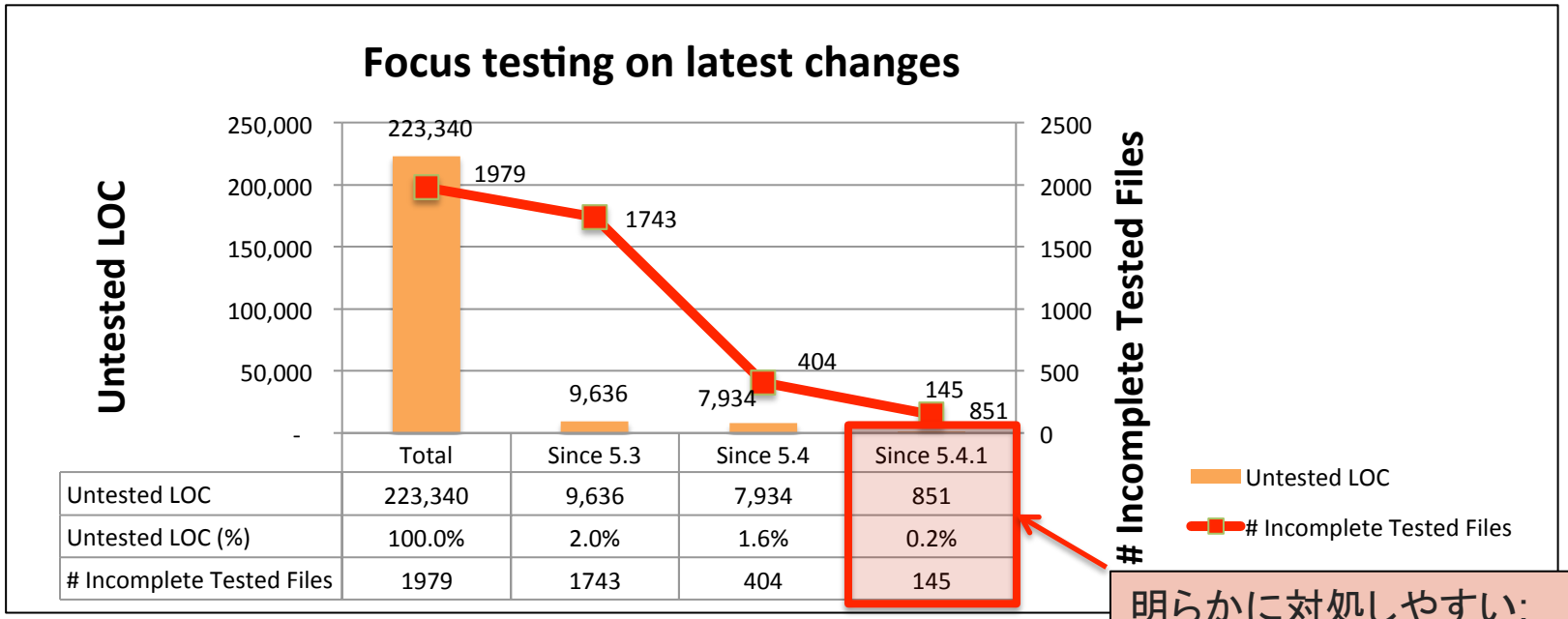
- どこからテストすべきか？
  - 以下のコードに着目した自動テスト開発をガイドする
    - 新規コード
    - リスクの高い部分を特に識別
    - 関連のないコードを除外
  - テストが不十分なコードを開発者に自動的に通知し、その対処を促す
  - アジャイルな受入れプロセスの一部として“テスト十分性”の施行を可能に

coverity®

実験

“チェンジセット”にフォーカスしたテストポリシーを評価する単純な試み

- フロントエンドと解析コード数：1,372,376 行
- 1,149,036 行がテストでカバー (83.7% カバレッジ), 223,340 行が未テスト
- 直前の変更にフォーカスすると、145 ファイル内の 851 行をカバーする必要がある

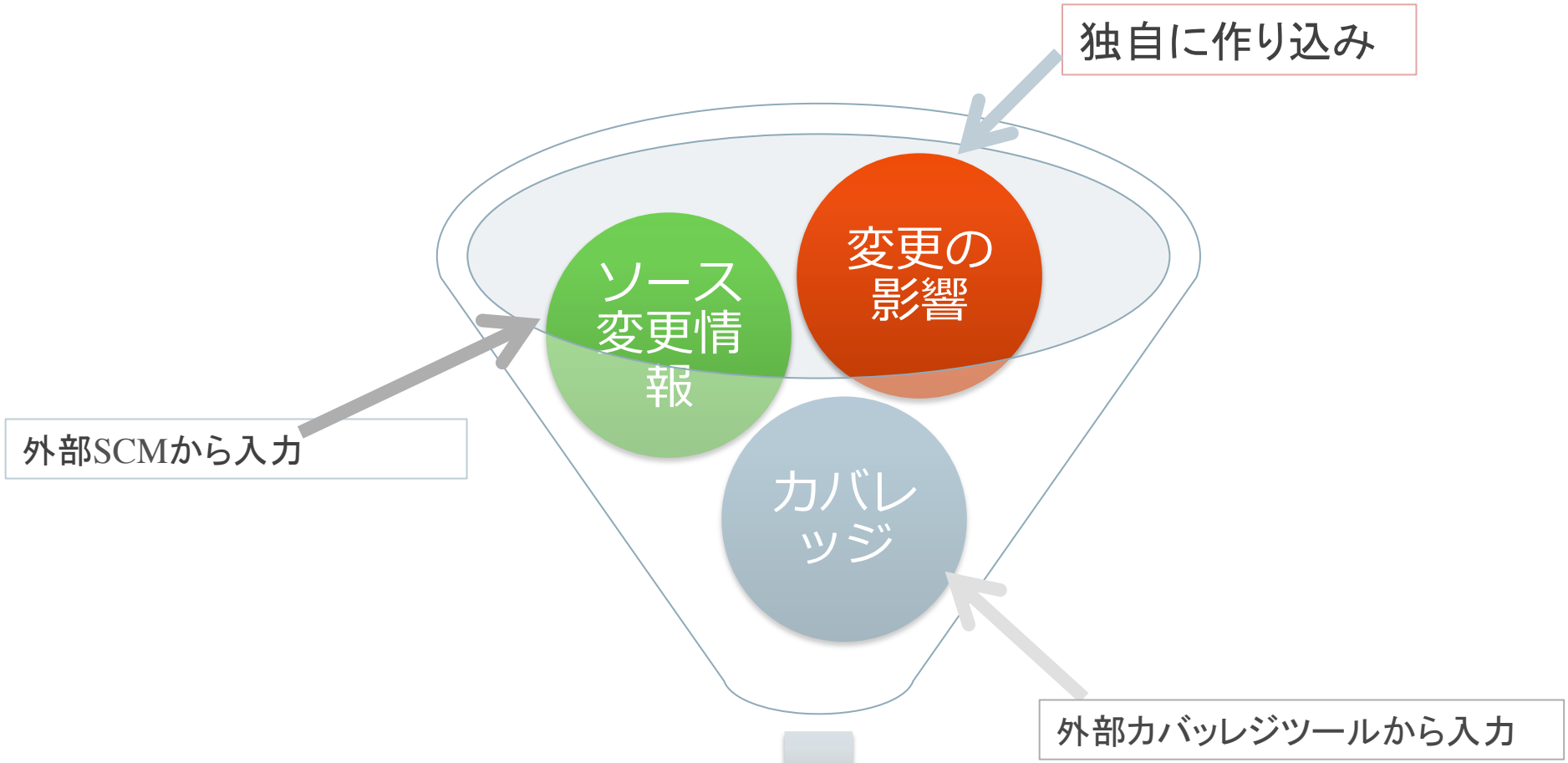


	Total	Since 5.3	Since 5.4	Since 5.4.1
Untested LOC	223,340	9,636	7,934	851
Untested LOC (%)	100.0%	2.0%	1.6%	0.2%
# Incomplete Tested Files	1979	1743	404	145

明らかに対処しやすい:
- 未テストコードの 0.2%
- 未テストファイルの 7%

coverity®

自動テスト開発支援ツールの実装

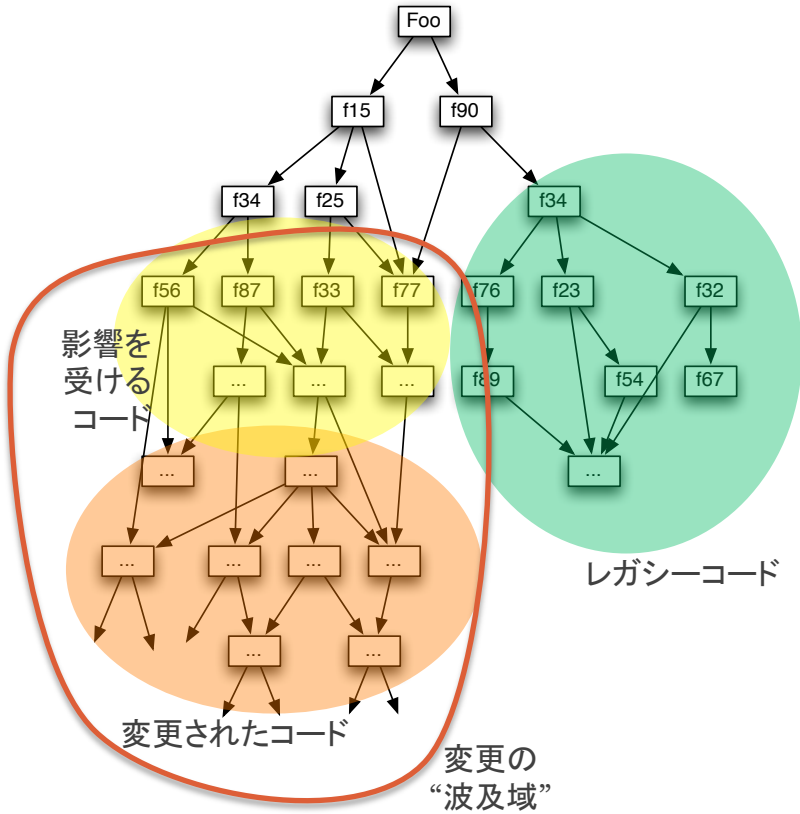


優先的にテストすべき箇所、
優先的に修正すべきテスト
を開発者に通知

coverity®

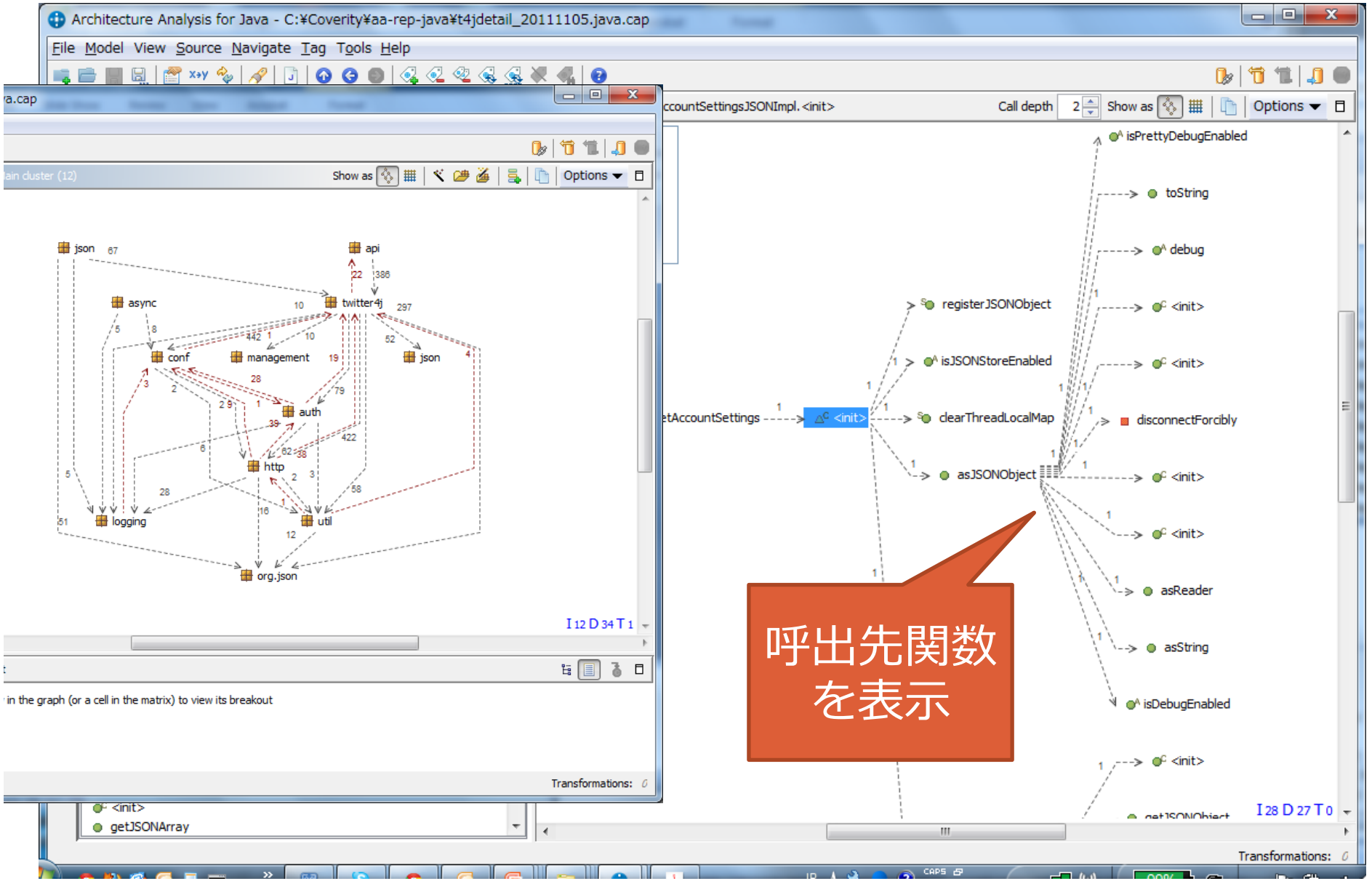
変更の影響解析

- チェンジインパクト解析は、変更があったコードが、他のコードに影響を与えるかどうかを的確に判断する。
  - 例 1 : 関数の挙動の変更は、その関数を呼び出すコードに影響を与える場合がある
  - 例 2 : クラスの継承関係の変更は、仮想関数呼出し時の実装決定に影響を与える場合がある

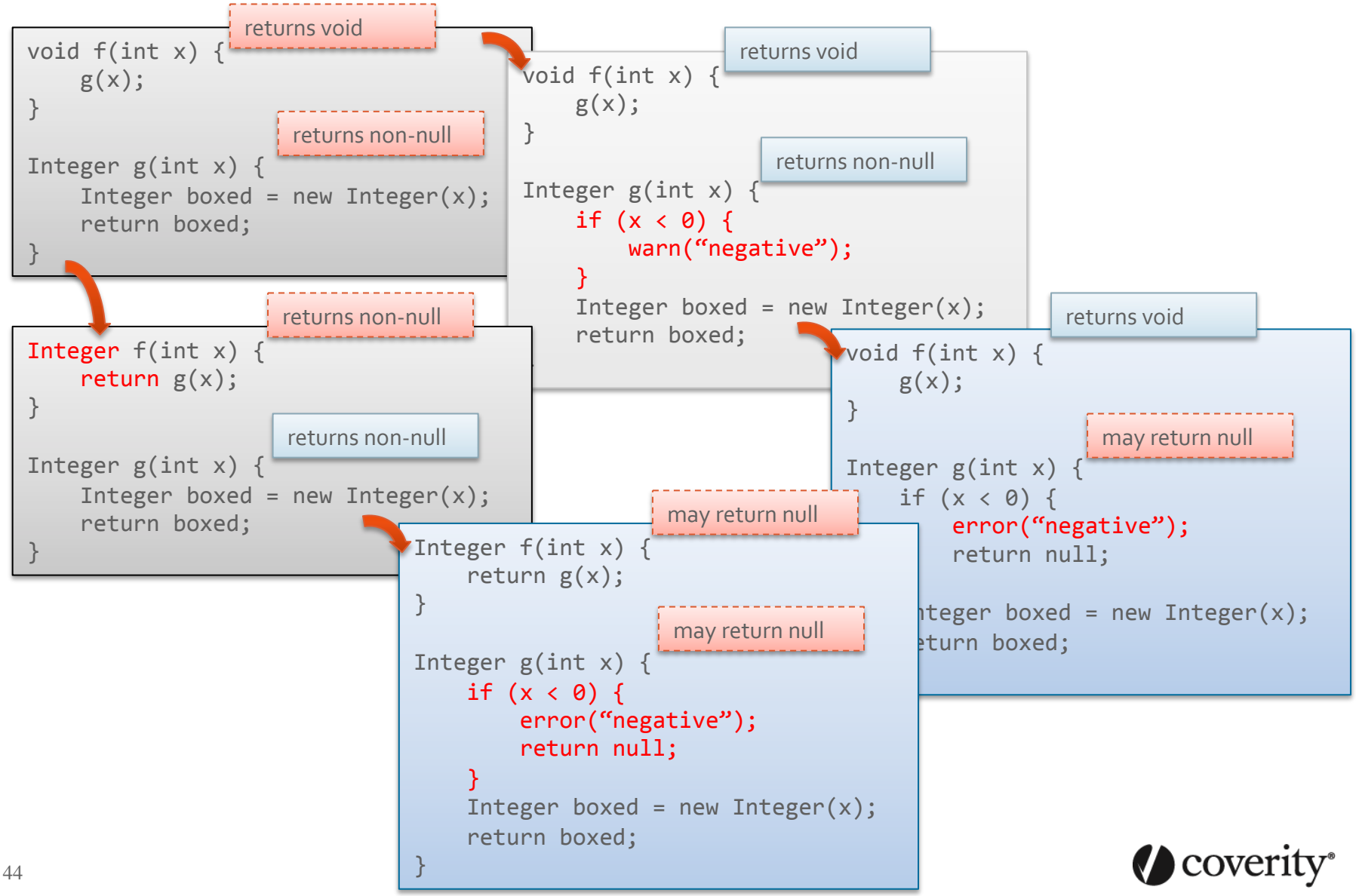


coverity®

これまでは構造的に見るのが主流



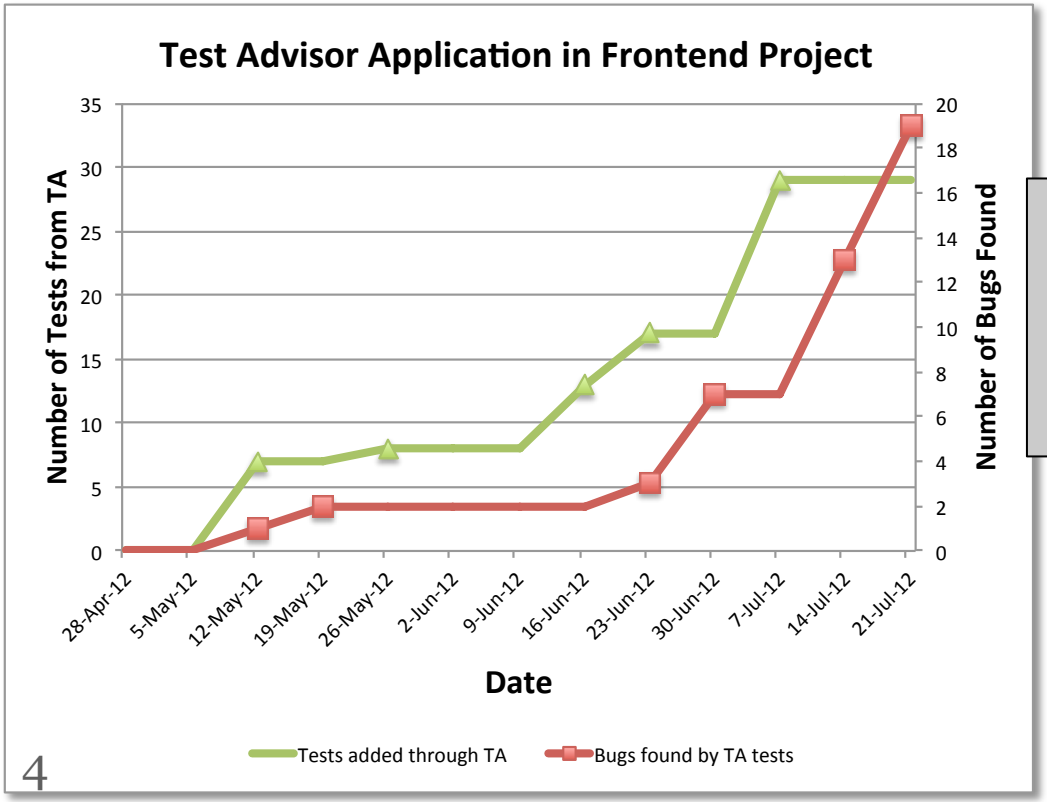
もう一つ先へ。変更の影響解析の例



coverity®

社内での経験 – ケーススタディ

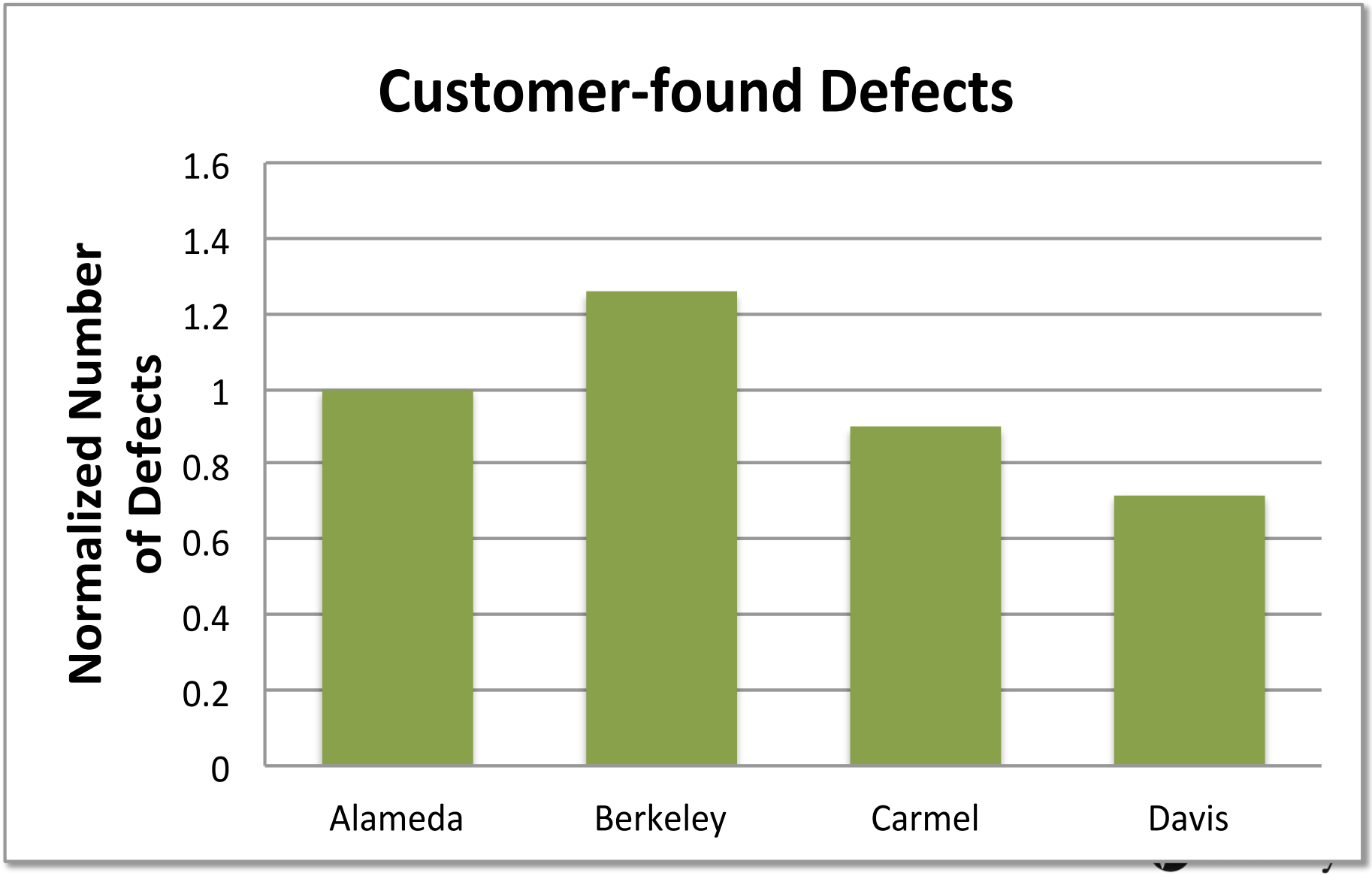
- 十分テストされている製品のコンポーネントにこの社内ツールを適用
  - 3ヶ月間、１テストエンジニアが１週間のうち４時間、テストすべきと判断された箇所にテストを追加した
  - テストポリシー: テスト可能なコードは 100% テストする (除外： デバッグコード, 実行されないコード, その他)



- １人週間換算で29件のテストを追加
- 製品から19件のバグを発見 – いくつかは重大なもの
  - 例： Keil コンパイラの正しい機能を阻害するバグなど

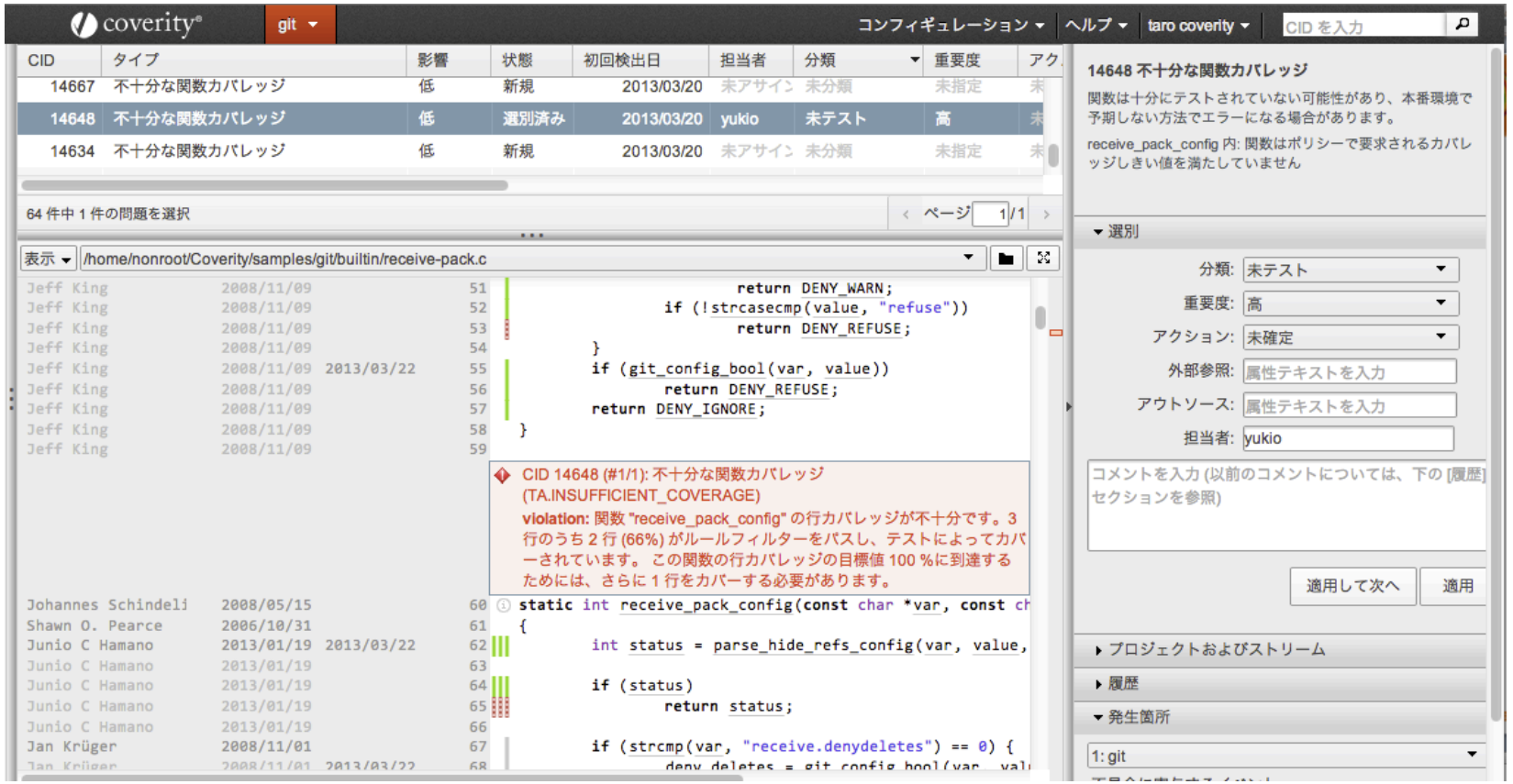
coverity

製品顧客が発見した不具合



Coverity 製品にもこの技術を組み込む

- Coverity Test Advisor　（Java，　C/C++, C#）
- 変更の影響箇所、高リスク箇所をお知らせします



インパクト解析を取り入れた テスト自動化ワークフロー

問題を適切な開発者に
アサイン

テスト作成とアップデート

- 結合レベルでの品質とセキュリティ脆弱性をチェック
- コードの追加・変更のため、新しいテストが必要かチェック

coverity®

まとめ

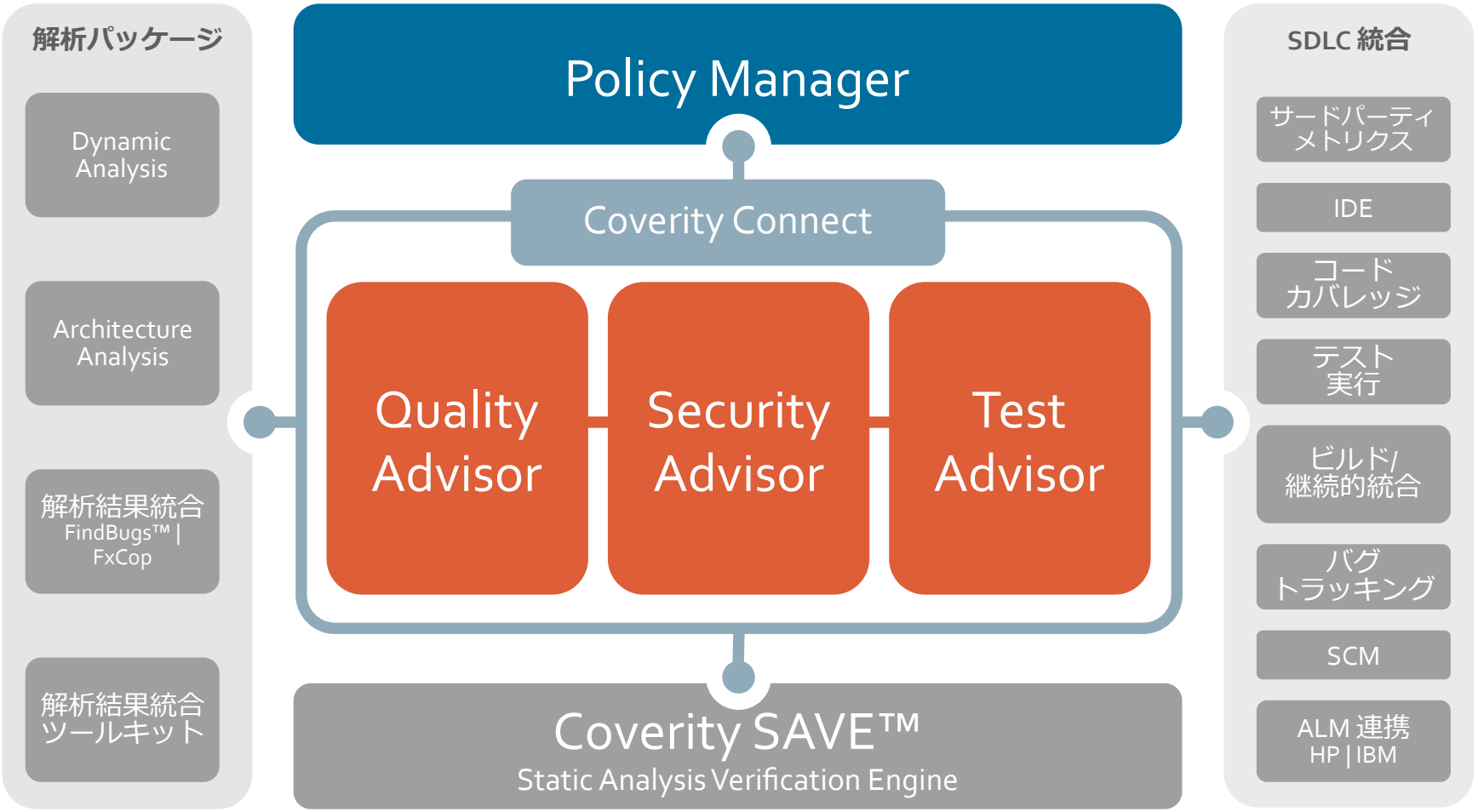
- No.5: テスト自動化が解決策？
- No.4: スキルセットは正しい？
- No.3: クリーンなコード?
- No.2: 正しい計測？
- No.1: 優先度を設定せよ！

Make every developer a rock star !

coverity

Coverity Development Testing Platform

解析 | 修正 | 統制



商用コード　|　オープンソースコード

coverity®