IBM

Javaコードのレビューノウハウ

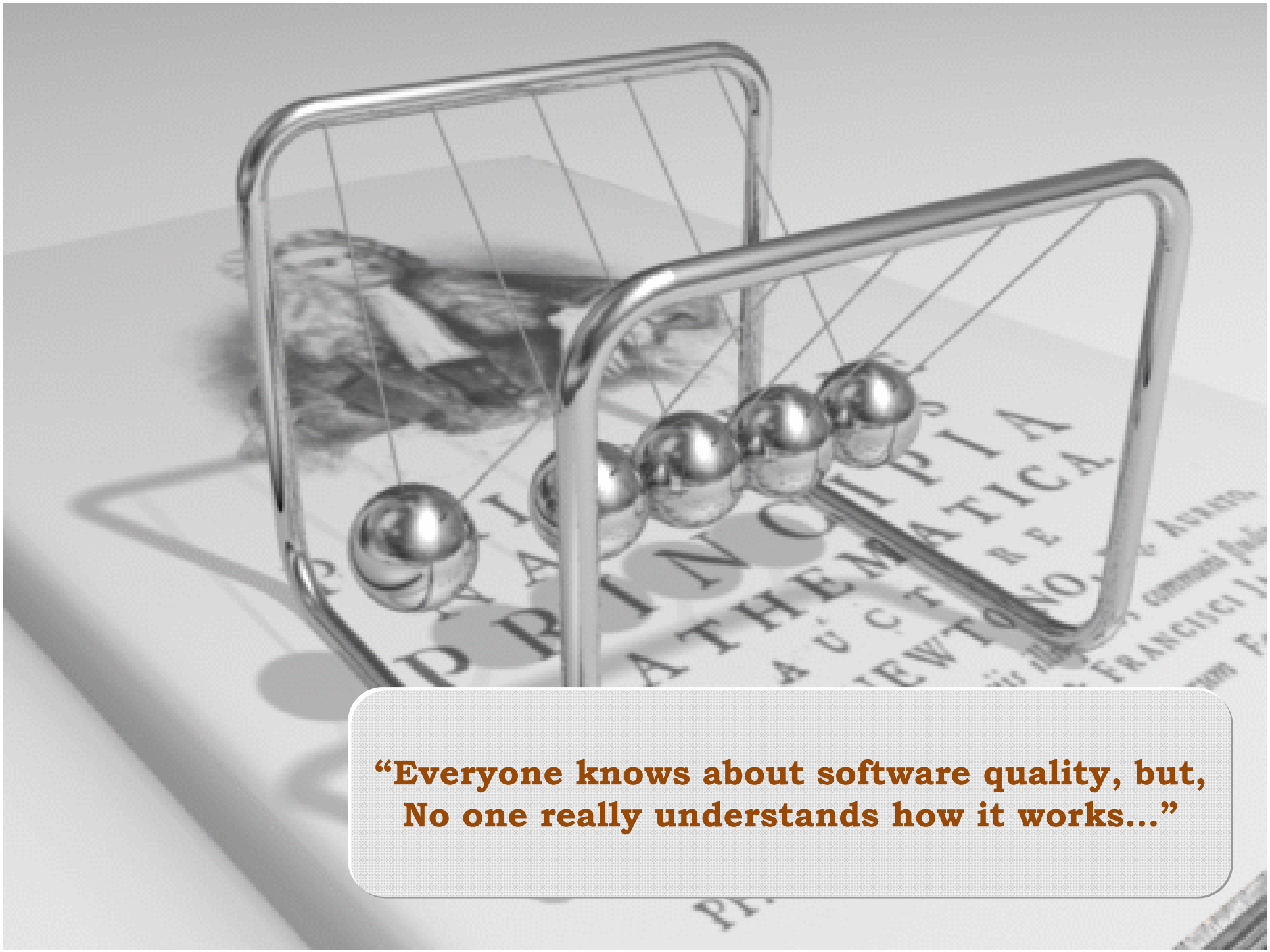
- IBM-QI法を用いたコードレビューの実践 -

Sapporo Java Connoisseurs Presents

JavaFesta2010 in 札幌

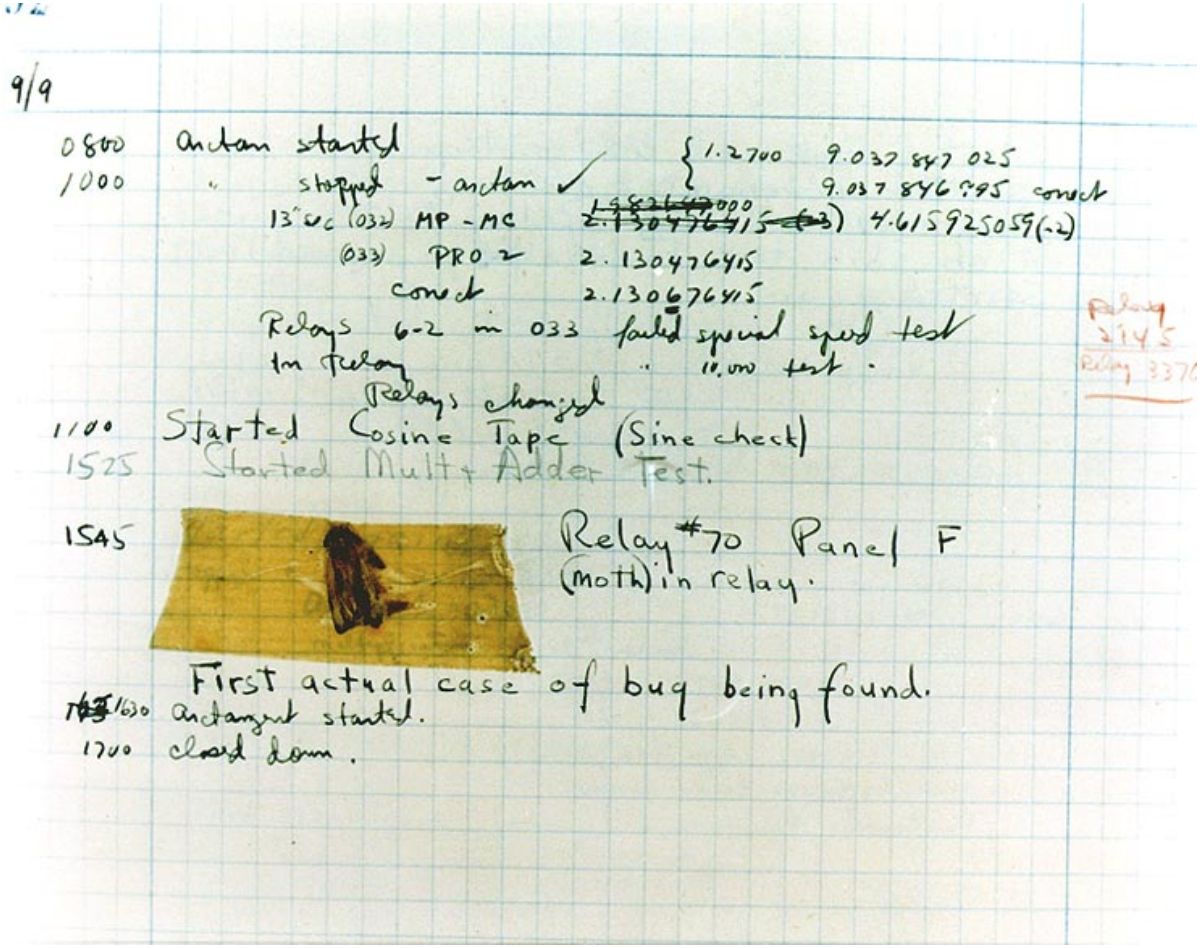
www.javafesta.jp

Nobuhiro Hosokawa (carvin@jp.ibm.com)
Manager
Quality Engineering, Services Quality.
IBM Japan Ltd.



**“Everyone knows about software quality, but,
No one really understands how it works…”**

世界で最初のバグ 1947年9月9日(1947-09-09)

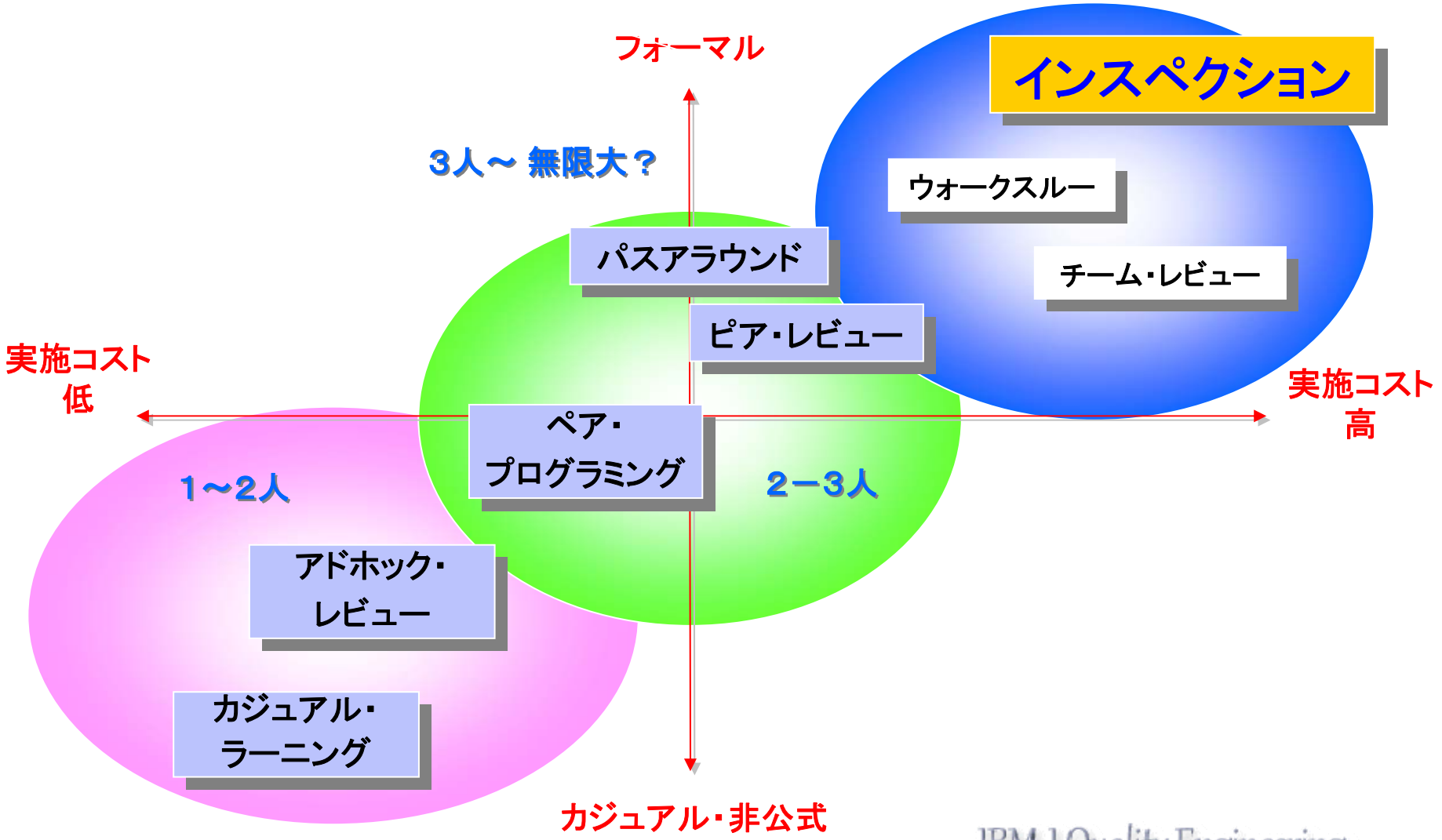


概要

The First "Computer Bug" Moth found trapped between points at Relay # 70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1947. The operators affixed the moth to the computer log, with the entry: "First actual case of bug being found". They put out the word that they had "debugged" the machine, thus introducing the term "debugging a computer program". In 1988, the log, with the moth still taped by the entry, was in the Naval Surface Warfare Center Computer Museum at Dahlgren, Virginia, which erroneously dated it 9 September *1945*. The Smithsonian Institute's National Museum of American History and other sources have the correct date of 9 September *1947* (Object ID: 1994.0191.01). The Harvard Mark II computer was not complete until the summer of 1947.

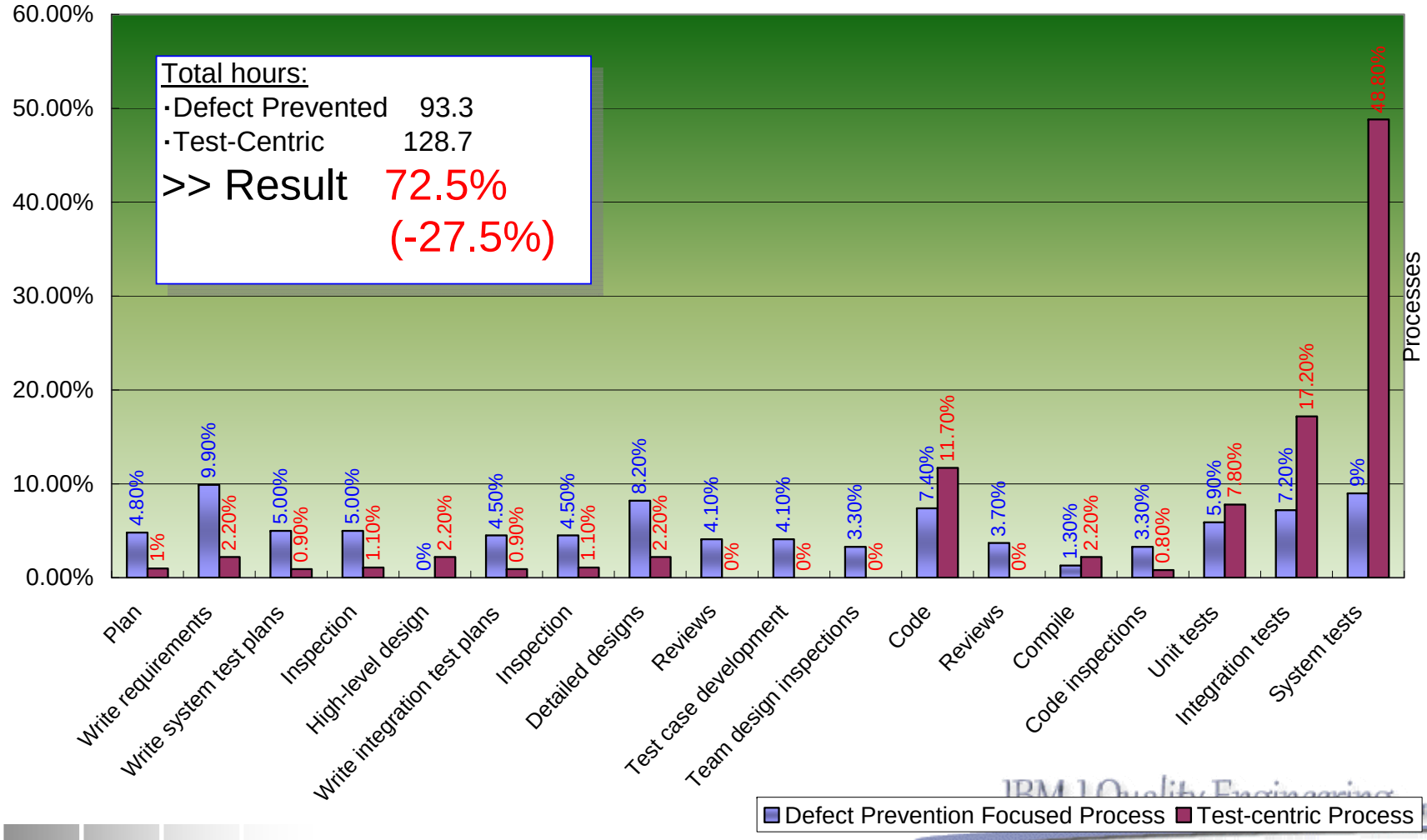
- Removed caption read: Photo # NH 96566-KB First Computer "Bug", 1945
- **日付** 1947**年**9**月**9**日**(1947-09-09)
- **原典** U.S. Naval Historical Center Online Library Photograph NH 96566-KN
- **著者** Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988.
- **許可** (このファイルの再利用) *This image is a work of a sailor or employee of the U.S. Navy,*
- *taken or made during the course of the person's official duties. As a work of the U.S. federal government*

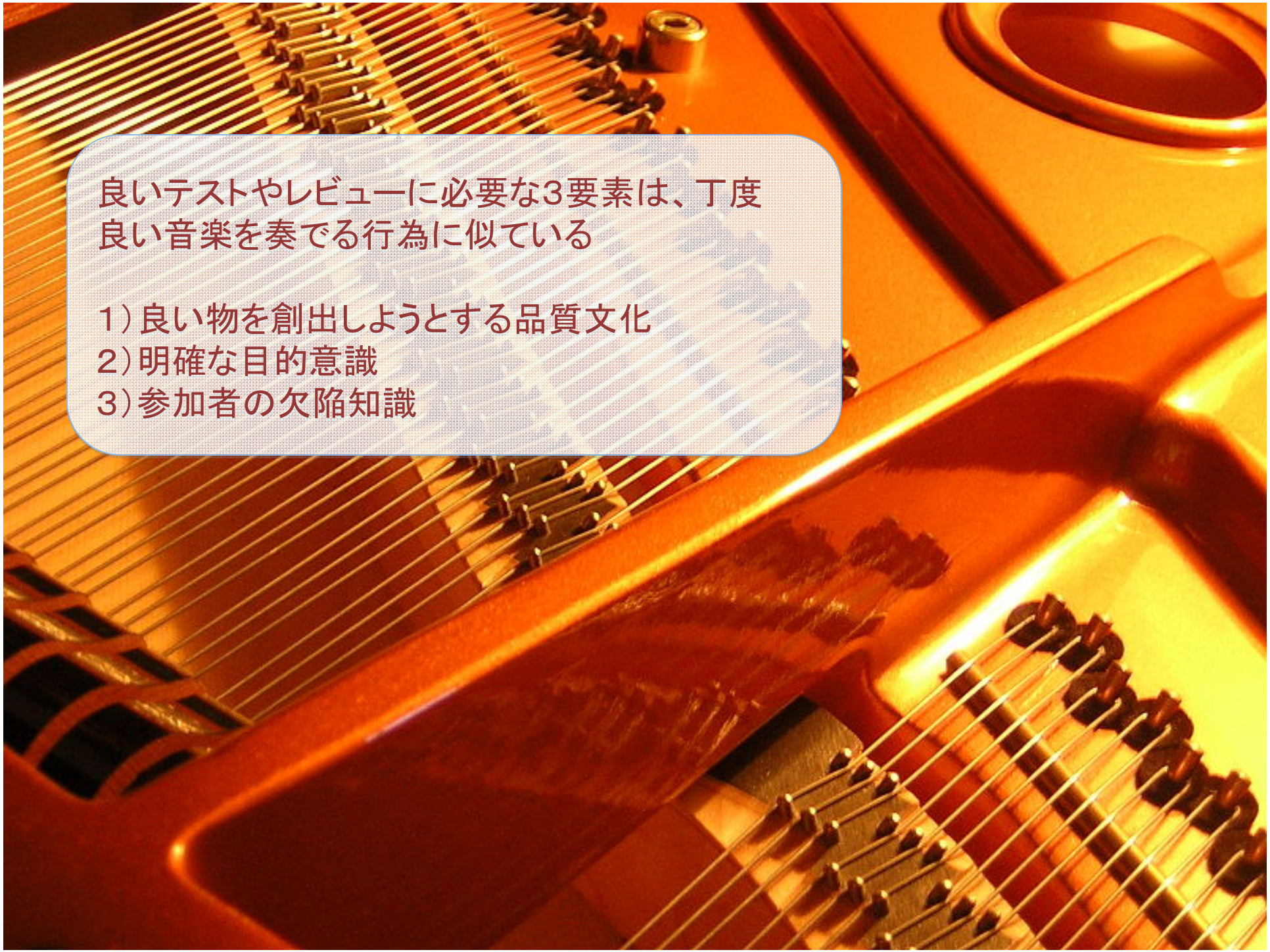
レビューの種別


フォーマル
インスペクション
3人～ 無限大？
ウォークスルー
パスアラウンド
チーム・レビュー
ピア・レビュー
実施コスト
低
実施コスト
高
ペア・
プログラミング
1～2人
2－3人
アドホック・
レビュー
カジュアル・
ラーニング
カジュアル・非公式

テスト中心プロセス vs 予防・予測中心プロセス

- 下記図の示すように、修正やテスト中心のプロセスよりも、品質に注力した予防・予測中心プロセスのほうが 約30%程度期間が短くなる事例が報告されています。




良いテストやレビューに必要な3要素は、丁度
良い音楽を奏でる行為に似ている
1）良い物を創出しようとする品質文化
2）明確な目的意識
3）参加者の欠陥知識

レビューの大敵:

- **プロジェクト制約:**
  - 予算制約・時間制約 → 丁寧に実施すると時間がかかる
  - 安易な商魂 → “早くリリースせよ”というプレッシャー
- **慢心・おごり・心理的思い込み**
  - 「自分だけは品質がよい。大丈夫だ」と思う感覚
  - レビューすれば必ず品質があがる
- **過去の悪習・ベテランのエゴ**
  - 「いつもこうやってきたから」

目視品質検査の重要性とIBM-Quality Inspectionの導入

■一般的にレビューやインスペクション等の人手による品質検査作業の重要性は広く認知されています。しかし、それらの多くは誤字脱字のみを検査する形式チェックにとどまる、労力に見合う効果が上がらない、検査そのものが属人的、結局すべての欠陥を除去しきれないといった多くの課題を内包します。

IBM Quality Inspection（品質インスペクション）とは？

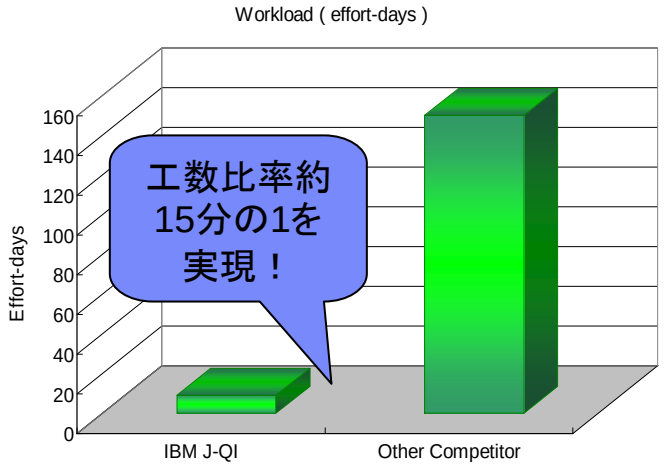
・2000年に日本アイ・ビー・エムのQA組織で開発された、プロジェクト外部の第三者による目視の品質検証サービスです。

・弊社が独自に開発した品質検証活動「QI: Quality Inspection」では、欠陥検出効率はほぼ従来のまま、品質検査工数を90%削減しプロジェクト全フェーズ中での頻繁な検査が可能です。

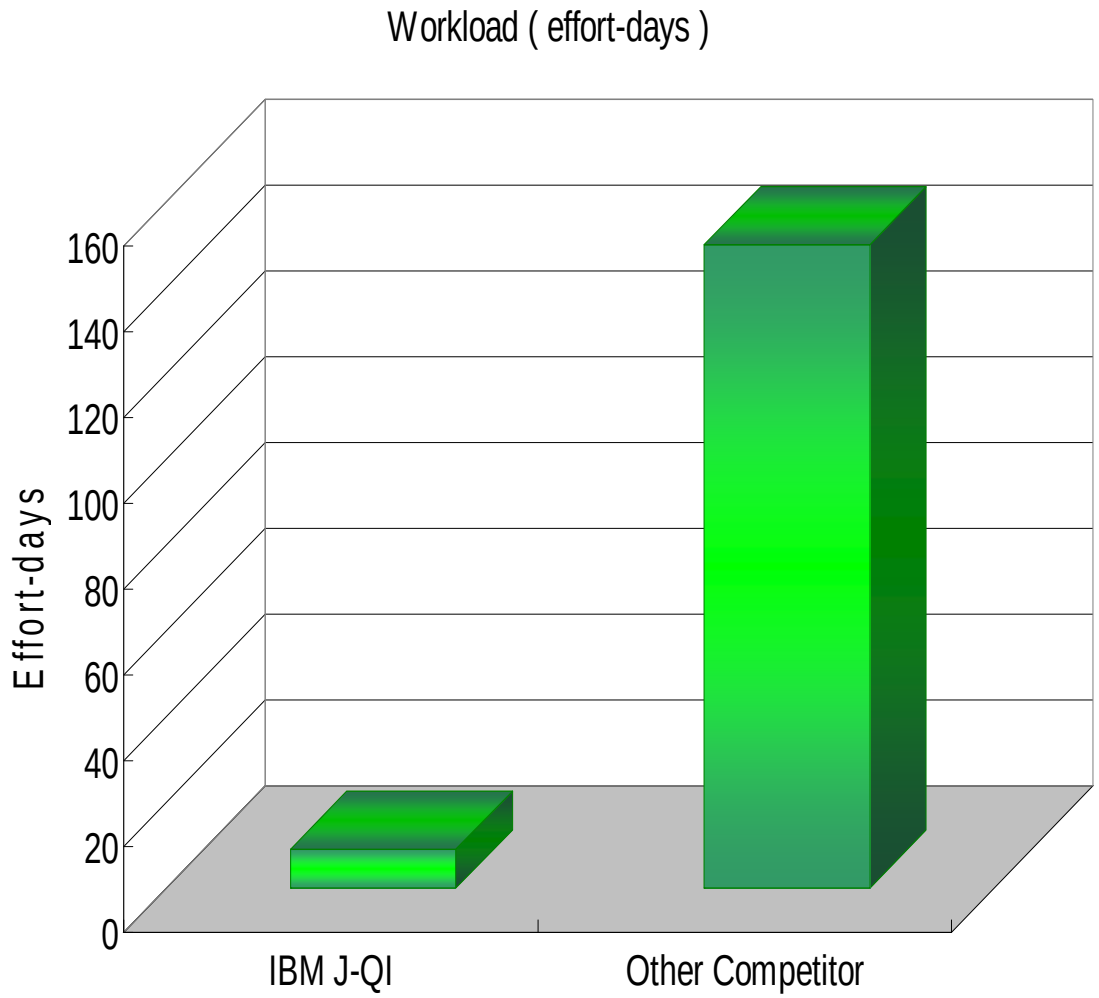
・QI手法によってプロジェクト成果物（仕様書・コード・テスト計画書その他）を目視による品質検査を実施することで

早期欠陥除去
品質状況可視化
品質のばらつき抑止

を実現し安定したプロジェクト進捗とコスト超過予防を実現します。



IBM-QI法の効果： 日本市場での比較



A) 所要工数
- 他社　150 人日
- IBM-QI法　9 人日

B) 検出欠陥の種類数
- 他社
  64 種類
- IBM-QI法
  62 種類

― 第2章 ―
欠陥の理解とバグの認識

欠陥の約60%～80%は、約20%のモジュールに存在する

- DOS/V OSにおける約500個のモジュールのうち、20%のモジュールでエラーの約80%を検出した（Endres 1975）
- Fault-prone性が高いと予測された上位20%のモジュールに、全フォールトの60%が含まれていた（Ohisson 1996）
- 再作業の80%は20%の欠陥によるものであった（Boehm 2000）
- 通信スイッチシステムに含まれる20%のモジュールに、60%のフォールトが含まれていた（Fenton 2000）
- 20%のモジュールに、運用時に発見されたフォールとの80%が含まれていた（Fenton 2000）
- 通信システムにおける欠陥の63%～70%は、20%のモジュールに含まれていた（Andersson 2007）
- GNU Compiler Collectionのプロジェクトで、80%のフォールトが20%のファイルに含まれていた（Hamill 2009）

欠陥がありそうな「約20%」モジュールを特定したい→fault-proneモジュール予測

1）人間には誤りを訂正して理解しようとする力がある

- **練習問題1　次の文章に間違いはいくつ？**

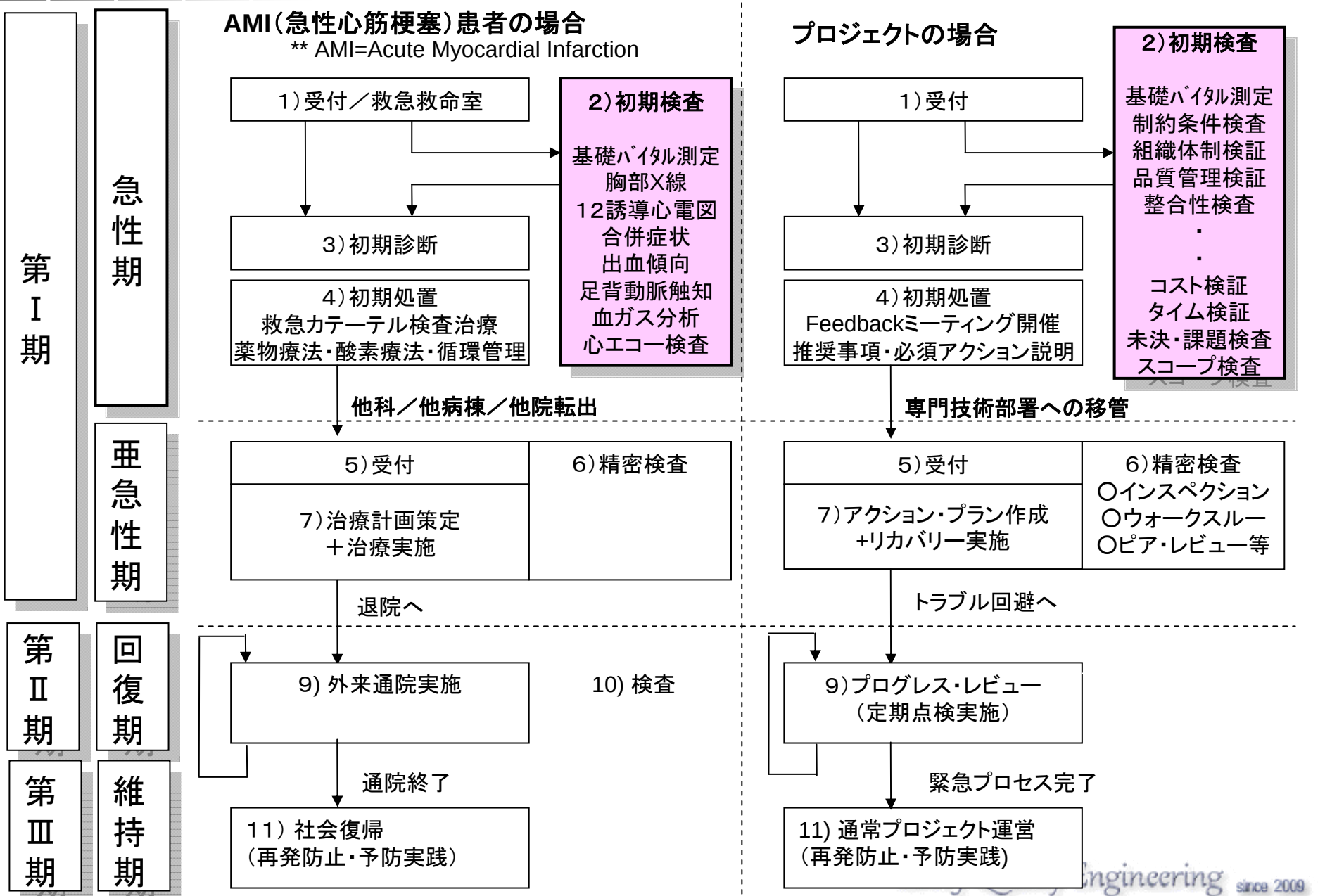
こんちには　みさなん　おんげき　ですか？ わしたは　げんきです

2）欠陥を予測する能力：品質バイタルサイン（QVS）

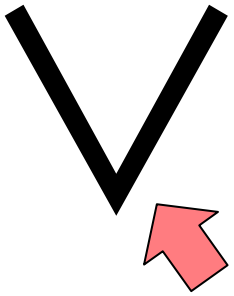
- **品質バイタルサイン** (QVS)とは?
  - 成果物の中身や意味を熟視テスト（Stare Test）実施前に，成果物のあらゆる**外的因子を測定し，加工して成果物検証作業の入力として利用する**. この時に測定する外的因子を「品質バイタルサイン」（以下QVS：Quality Vital Signs）と呼ぶ.

- プログラムによる機械的な抽出 → グラフ等による視覚化
- パターン分析による推奨事項・リスクアセス

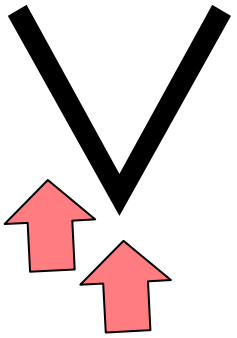
【効果】
- **品質検証時間／TATの短縮**
- 属人性の排除+品質検証コストの削減
- 欠陥検出活動の精度向上


第Ⅰ期
急性期
亜急性期
第Ⅱ期
回復期
第Ⅲ期
維持期
AMI（急性心筋梗塞）患者の場合
** AMI=Acute Myocardial Infarction
1）受付／救急救命室
2）初期検査
基礎ﾊﾞｲﾀﾙ測定
胸部X線
12誘導心電図
合併症状
出血傾向
足背動脈触知
血ガス分析
心エコー検査
3）初期診断
4）初期処置
救急カテーテル検査治療
薬物療法・酸素療法・循環管理
他科／他病棟／他院転出
5）受付
6）精密検査
7）治療計画策定
＋治療実施
退院へ
9) 外来通院実施
10) 検査
通院終了
11）社会復帰
（再発防止・予防実践）
プロジェクトの場合
1）受付
2）初期検査
基礎ﾊﾞｲﾀﾙ測定
制約条件検査
組織体制検証
品質管理検証
整合性検査
・
・
コスト検証
タイム検証
未決・課題検査
スコープ検査
3）初期診断
4）初期処置
Feedbackミーティング開催
推奨事項・必須アクション説明
専門技術部署への移管
5）受付
6）精密検査
〇インスペクション
〇ウォークスルー
〇ピア・レビュー等
7）アクション・プラン作成
+リカバリー実施
トラブル回避へ
9）プログレス・レビュー
（定期点検実施）
緊急プロセス完了
11) 通常プロジェクト運営
（再発防止・予防実践)

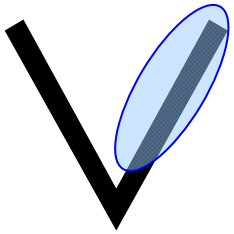
3）闇雲にレビューしない：アタリをつける技術



V字モデルの、“この辺”
の時点で、

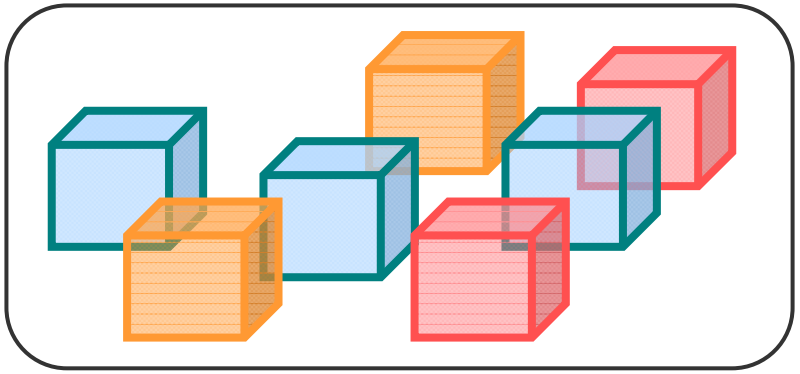


V字モデルの、“この辺”
の時点で測定可能なプロダクトメトリクスを用いて、

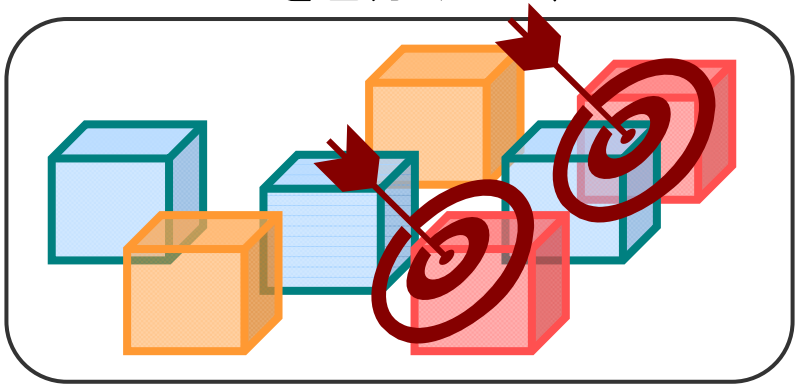


V字モデルの、“この辺”
で欠陥が見つかりそうな
モジュールを、

※ただし、あくまで「統計的傾向」に基づく判断である。
「AならばBである」のように、絶対的な結論は出せない。

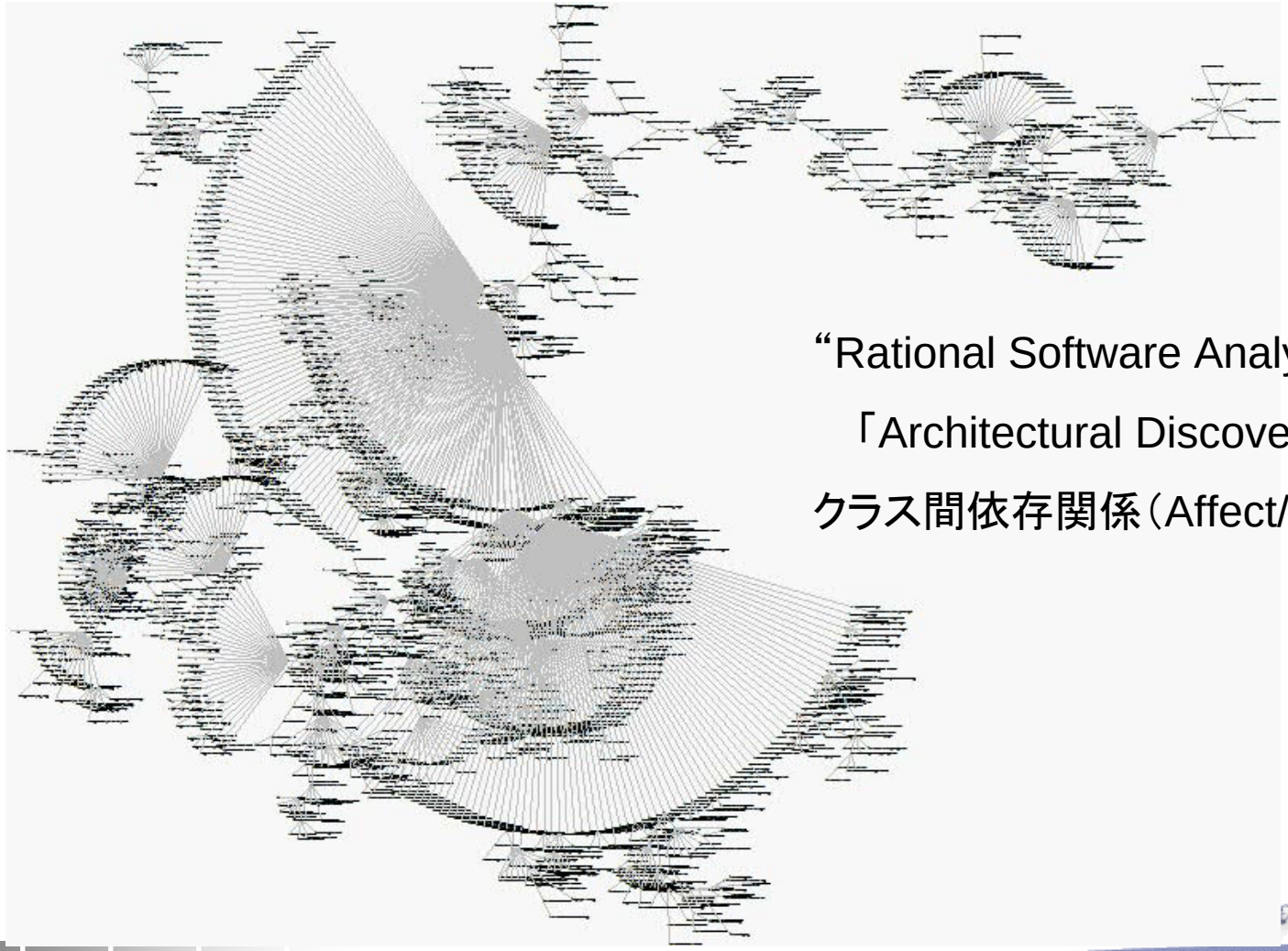


欠陥がありそうな度合いによって
モジュールを色分けして、



優先的／重点的にレビューしたりテストしたりすることで、レビューやテストの効率／効果を高めたい

例）クラス間の依存性から設計課題を「俯瞰」



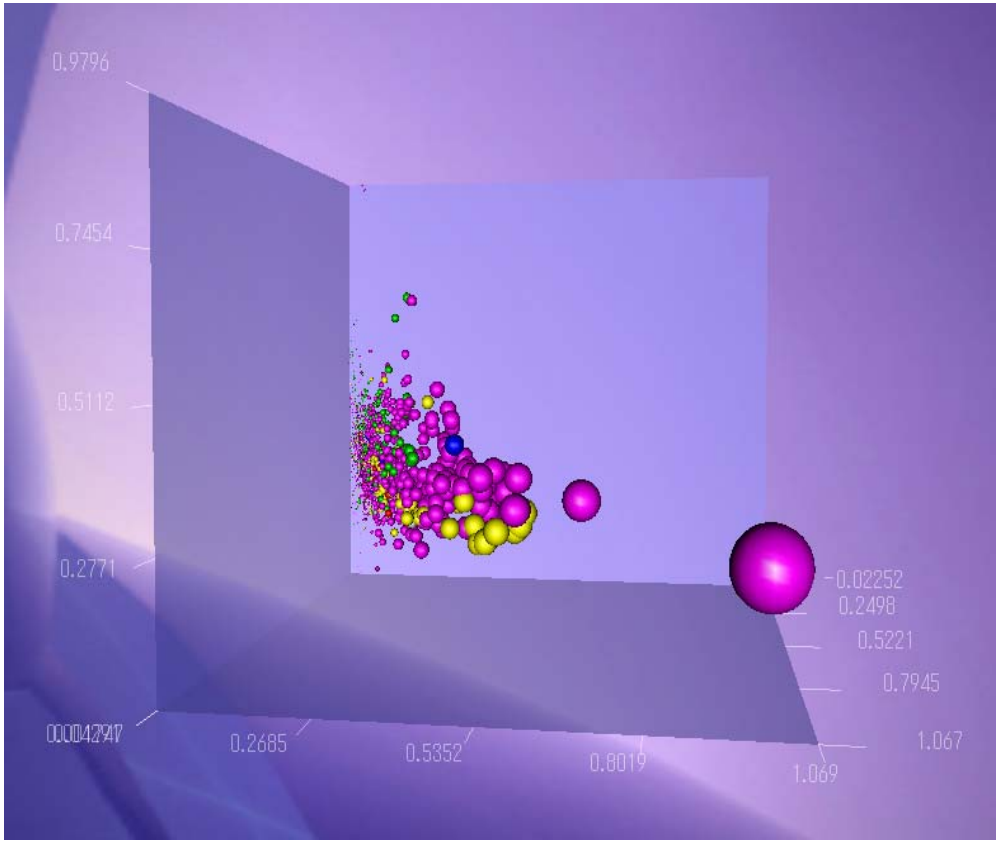
“Rational Software Analyzer”ツール出力

「Architectural Discovery」機能による

クラス間依存関係（Affect/Effect）の可視化

3D-Plot : Visualizing tool for project quality status

- Plot metrics to visualize quality
  - X-axis: “Size Factor” : LOC
  - Y-axis: “Skill Dependency” : Comment Rate
  - Z-axis: “Complexity” : # of IF
  - Color: Sub system groups
- Findings :
  - Found 1 outlier of size metrics
  - Comment rate is wide distributed
  - “Green” is size controlled well.
  - “Yellow” has less comment and complex
    - >> Weak control and management
- Recommended Action :
  - List outliers on 3 aspects and monitor them
  - Re-Design largest size program
  - Add comment to small size source.



― 第3章 ―
コードリーディング技術

コードリーディングの技術

- **早く読むための技術**
  - レビューを実施する際に、なるべく早くコード作成者の意図を汲み取り、設計基準を把握した方が、時間効率／欠陥検出効果が高まります
  - コードの目視レビュー時には、ソースコードを「文字」「文字配列」「複数処理ブロックの塊」「日本語（コメント）のみの集まり」「制御フロー構造」として捕らえる等様々に視点を変えて全体をすばやく把握することが重要です。
  - 最も欠陥検出率の高い目視レビューが実施できる限られた時間の中で、いかに「目視サンプリング率」を高めるかが全体品質を高める鍵を握ります。
  - 次ページ以降に示した技術はコードリーディングの効率化と、より大局的なコード俯瞰のための技術です。どのような場面でもコードに関わるあらゆる担当者にとって有用な技術を中心にお伝えします。

早く読むための技術：コードはお尻から読む

原則	コードはお尻から読む
理由	コードが修正される際に関数やメソッドの追加はファイルの末尾に追加される場合が多いため、末尾に追加された関数だけ「元の担当者でない」書式をする場合がある
実害	▪ 過去の経緯、元のコードの意図を十分に理解しない拙速なコード修正が入り込みやすい ▪ 複数人数による開発でテスト容易性（＝試験性の一部）が低下する

例：Javaの例）

```
public class AAAAAImpl implements BBBBB {
        private Jtc31_koutei_calendarDAO c31Dao;
        public Date getShoriKijunDate(Date nyuryokuNengappiDate, String syukeiTaniFlag, String syoriType){
            :
        }
        private Date getShoriKijunDateByNitiji(Date nyuryokuNengappiDate, String syoriType){
            :
        }

        public Date previousMonday(Date nyuryokuNengappiDate) {
                Calendar c = Calendar.getInstance();
                    :
                c.add(Calendar.DATE, -7);
                return c.getTime();
        }
}
```

←publicメソッド？

getXXXメソッド命名違反

「-7」という定数直書き

考慮点	▪N/A

早く読むための技術：全体俯瞰＝ヘッダーコメント

原則	全体俯瞰＝まずヘッダーコメントの有無を確認し、改定履歴を確認する
理由	ヘッダーコメントが記載されていないコードは①作成途中、②プロではない作成者、③急いで作った、④テストクラスである場合、等の可能性がある。
実害	▪ 改定履歴がないため、保守担当者に改修の難所が判別しにくい（保守性・理解容易性） ▪ 処理概要からクラス／コードの「アイデンティティ」が判別しにくいため、結果として構造化崩しや異なる責務を実装し、コードの設計を崩しやすい（短寿命・保守改修劣化が早い）

例：

```
/*
 * ｘｘｘｘプロジェクト
 * yyyy工程管理
 * (c) Copyright zzzzzz Ltd. 2009 All rights reserved.
 *
 * version 1.0
 * Date 2009/07/07
 */
```

←JavaDocルール?

クラスのヘッダーコメント？

Dateは作成日？更新日？

考慮点	▪N/A

早く読むための技術：ファイル名・変数名に着目

原則	ファイル名、変数名から、どれくらいの年齢の人が書いたか判定する
理由	ハイフンやアンダースコアの使い方に着目すると、汎用機言語、C言語、Java他の言語、VBといった開発者の言語背景が理解できる。犯しがちな誤りが判別できる。
実害	▪ 過去の経緯、元のコードの意図を十分に理解しない拙速なコード修正が入り込みやすい ▪ 複数人数による開発でテスト容易性（＝試験性の一部）が低下する場合がある

例：以下は全てJavaの例）

COBOL風	C/C++風	Java風
CALC-DATE-URU.java	Date_Leap_Year.java	LeapYear.Java
↑・TryCatch文を疑う ・関数長・長いメソッド	↑・NullPointer評価の正当性 ・メモリ処理・例外処理	↑・newやsyncronized キャスト

考慮点	▪N/A

参考）予測のための指標例：平均変数名長

- メトリクス名：平均変数名長
- 英語名：　Average Identifier Length
- 単位：　1ソースコードあたり平均値1つ
- カテゴリ：　属人性指標
- 取得方法：　1ソースコードに記述された全て変数名の長さ（＝文字数）の平均値（単純総和平均）
- 備考：

当該ソースコードの開発者がどのような開発言語経験を持つか推定することができる

予測のための指標例：平均変数名長をどう使うか？

1）以下の判定条件式にしたがって、開発者の言語経験を推定する

IF	平均変数名長≦8	THEN	COBOL言語経験者
IF	9≦平均変数名長≦12	THEN	C/C++言語経験者
IF	平均変数名長≧18	THEN	JAVA言語のみの経験者

2）当プロジェクトでの使用言語と1）の経験言語を照合し混入欠陥を予測する

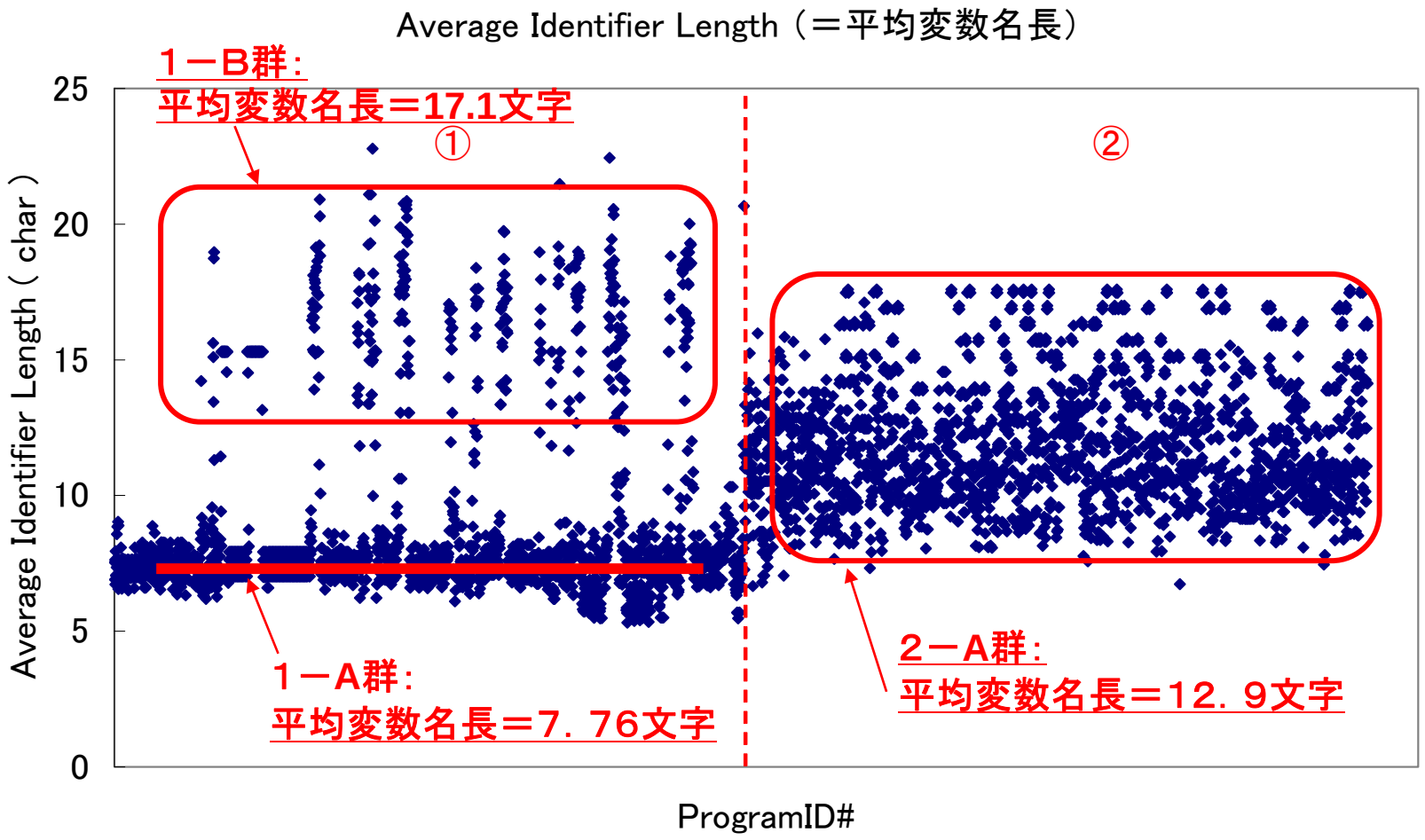
今回利用言語が「Java」で、

開発経験言語がCOBOLなら、「例外処理の漏れ／try-catch」を疑う

開発経験言語がC・C++なら、「Null Pointer エラー類」を疑う

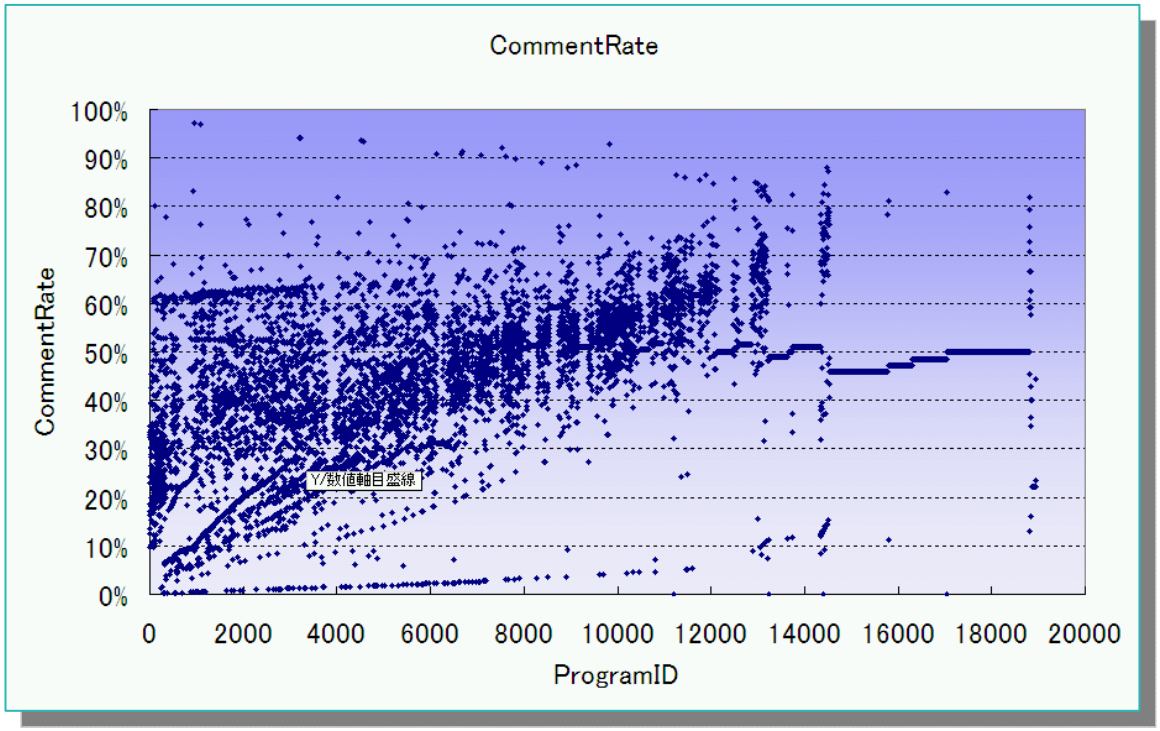
開発経験言語がJavaなら、「リソース解放忘れ／new多用」を疑う

平均変数名長(AIL)


Average Identifier Length（＝平均変数名長）
1－B群:
平均変数名長＝17.1文字
①
②
25
20
15
10
5
0
Average Identifier Length (char)
1－A群:
平均変数名長＝7．76文字
2－A群:
平均変数名長＝12．9文字
ProgramID#

早く読むための技術：メトリクスに着目する

- コード・メトリクスの測定＝ある意味での「コードの抽象化・俯瞰」技術
- 特定の目的に向けたコードメトリクスは、コードリーディング作業前に測定しておくことで読み手に「良い意味で固定観念」を与える



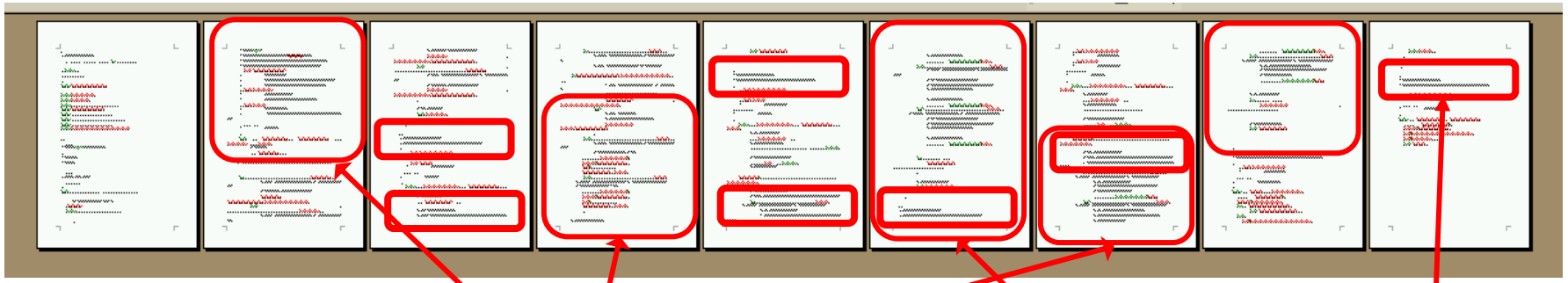
1プロジェクト分のコメント率散布図

① [事実の洞察]
- 100%のコメント率がある
- 帯状分散のコメント率傾向がある（ばらついている）
- クローン形跡がある
- コメント率10%以下が左端にある
- 数が多い

② [仮説]
- コードが均質でない可能性がある
- コーディング標準にコメントに関する記載・ガイドがない可能性がある
- **頻繁な仕様変更（改変・変更要求）が加えられた可能性がある**
- 協力会社が偽った可能性がある

全体を俯瞰するための技術：コードをズームアウトする

- Microsoft Word等にソースコードを貼り付け、「ズームアウト」表示すると全体の制御構造が俯瞰しやすく、どこからコードを見るべきかが把握しやすい



SampleJavaのWord表示（倍率12%ズームアウト）例

検査観点：

1. 黒いところ＝処理が複雑で重たい可能性が高い
2. 右にへこんだところ＝ネストが深い構造・複雑な分岐やループの可能性
3. メソッドの塊は見えるか？＝メソッドコメントの存在とその統一を観察

全体を俯瞰するための技術：コードをソートする

- コードを各行の文字数で降順ソートすると、コピー＆ペースト状況や処理の抜けを検出できる場合がある

```
public class CJPR110FCNImpl implements CJPR110FCN {
   /** 工程機能カレンダー DAO */
    private Jtc31_koutei_calendarDAO c31Dao;
     /**
      * 処理基準日取得
      * 実績管理画面、作業状況管理画面で呼び出すのメソッド
      * 入力パラメータ.集計単位フラグの値を元に後続処理の判断を行う。
      * 入力パラメータ.処理タイプの値を元に処理基準日を算出する。
      * @param nyuryokuNengappiDate
      *                入力年月日Date
      *                業務共通の共通部品を用いて取得したオンライン日付、
      *                もしくは、セッションから取得した前処理基準日
      * @param syukeiTaniFlag
      *              集計単位フラグString
      *              0：日次処理、1：週次処理、2：月次処理
      * @param syoriType
      *              処理タイプString
      *              0：当日処理、1：前週（月／年）処理、2：次週（月／年）処理
      * @return Date
      *              処理基準日
      */
                  :
                  :
```

降順ソート

```
// 目的：工程機能カレンダーを元に次週（月／年）の
// 目的：工程機能カレンダーを元に次週（月／年）の
// 年度初め開始日を算出する。
// 日次の処理基準日
// 点検実施年月の開始日を算出する。
// 点検実施年月の開始日を算出する。
// 処理基準日を算出する。
// 処理基準日を算出する。
// 処理基準日=入力年月日の前週の月曜日
// 処理基準日=入力年月日の週の月曜日
                  :
                  :
// 2-3の処理へ 週次の処理基準日を取得
// 2-3-1-2.  入力パラメータ.処理タイプ ＝ '1'（前週（月／年））の場合
// 2-3-1.【入力パラメータ.処理タイプの値によって以下の処理を分岐】
// 2-3. 週次の処理基準日を取得
// 2-2の処理へ 日次の処理基準日を取得
// 2-2-1-2.  入力パラメータ.処理タイプ ＝ '1'（前週（月／年））の場合
// 2-2-1-1.  入力パラメータ.処理タイプ ＝ '0'（当日処理）の場合
// 2-2-1.【入力パラメータ.処理タイプの値によって以下の処理を分岐】
// 2-2. 日次の処理基準日を取得
// 2-1-3.  入力パラメータ.集計単位フラグ ＝ '2'（月次処理）の場合
// 2-1-2.  入力パラメータ.集計単位フラグ ＝ '1'（週次処理）の場合
// 2-1-1.  入力パラメータ.集計単位フラグ ＝ '0'（日次処理）の場合
```

- ソースをソートした結果からは、①複数行に渡ってコピーした形跡、②コメントレベルで条件分岐が漏れている可能性、③ロールバリュー値が正しいか不明といった観点でレビュー開始候補が特定できる

全体を俯瞰するための技術：関数名称だけを読む

- 最後に追加した関数、およびオリジナル作成者と異なる改修者が作成した関数が分かると、セマンティックエラーや”しつこい”欠陥の原因に突き当たることがある。（特にLOCサイズの大きなコードに有効）

Sample2.PLI

LOC: 38674行

Size:1689KB

関数名のみ取得

Grep/Find機能で

「: PROC;」を検索

```
INIT_RTN: PROC;
INIT_RTN: PROC;
INIT_RTN: PROC;
INIT_RTN: PROC;
JYOKEN_HANTEI_RTN: PROC;
JYOKEN_MEISAI_RTN: PROC;
KAIIN_NO_SET_RTN: PROC;
KAISU_HANTEI_RTN: PROC;
KEIDKN_A_TO_A_RTN: PROC;
KEIDKN_A_TO_A_RTN: PROC;
KEIDKN_A_TO_B_RTN: PROC;
KEIDKN_A_TO_B_RTN: PROC;
KEIDKN_B_TO_A_RTN: PROC;
KEIDKN_B_TO_A_RTN: PROC;
KEIDKN_B_TO_B_RTN: PROC;
KEIDKN_B_TO_B_RTN: PROC;
KEIHIN_DOUKON_RTN: PROC;
KEIHIN_DOUKON_RTN: PROC;
LAST_RTN : PROC;
LAST_RTN : PROC;
LAST_RTN : PROC;
LAST_RTN : PROC;
LAST_RTN : PROC;
LAST_RTN : PROC;
LAST_RTN: PROC;
LAST_RTN: PROC;
LAST_RTN: PROC;
```

- 719関数!
- LAST_RTN=46箇所
- コピペ形跡（半角ｽﾞﾚ）
  1. 「LAST_RTN : PROC; 」
  2. 「LAST_RTN: PROC; 」
- 大人数での修正形跡
- 修正履歴なし

全体を俯瞰するための技術：コメントの記載クセを読む

- コメント文中の「。」や「、」の入り方に着目すると、複数人数で作成・修正を行っている箇所が分かる

例１）機能名と機能説明内容が「。」と「、」が同じか？

チェック箇所１）

- Mdl_UZ05A50_01.bas(322): ‘ 機能　　:ログイン情報引継パラメタを取得し、メニュー画面を表示する。
- Mdl_UZ05A50_01.bas(326): ‘ 機能説明　:共通関数よりログイン情報引継パラメタを取得し、画面を表示する。

チェック箇所２）

- Mdl_UZ05A50_01.bas(3862): ‘ 機能　　:CAT本設置連携訂正登録画面遷移
- Mdl_UZ05A50_01.bas(3869): ‘ 機能説明　:パラメタをセットし、未連携一覧画面への遷移を行う。

チェック箇所３）

- Mdl_UZ05A50_01.bas(4288): ‘ 機能　　:パラメータ振り分け関数
- Mdl_UZ05A50_01.bas(4298): ‘ 機能説明　:パラメータ（請求先番号）により、制御番号を返す

合計３名の印象。コードの末尾（４５９４Step）に近い方から、「最後に追加された可能性」を検証する


いかなる欠陥もまた人工物なり
(All defects are also “Artifact”)