JaSST’10　Tokyo
テクニカルセッション

オフショアコードの見える化事例

～アーキテクチャ分析支援ツールを使って、言葉の壁を乗り越える～

2010年 1月28日

ビースラッシュ(株)　酒井郁子

※ 本事例は、組込みプレス次号
(Vol.18)にも、掲載予定です。

オフショア開発で、
こんな問題に直面したことはないですか？

- コミュニケーションが取りにくい
  - 仕様のとり違い、変更の説明が困難
  - 品質要求に対する、認識の違い

→ 予定外の手戻り工数増加
開発スケジュールの遅延に

- 品質確保は、最後の最後になる
  - ブラックボックスによる、システムテスト
  - 委託先の提出する、動作確認書

→ 問題が発覚すると
再びシステムテストは待ちに
テストの遅延が、
そのままリリースの遅れに・・・

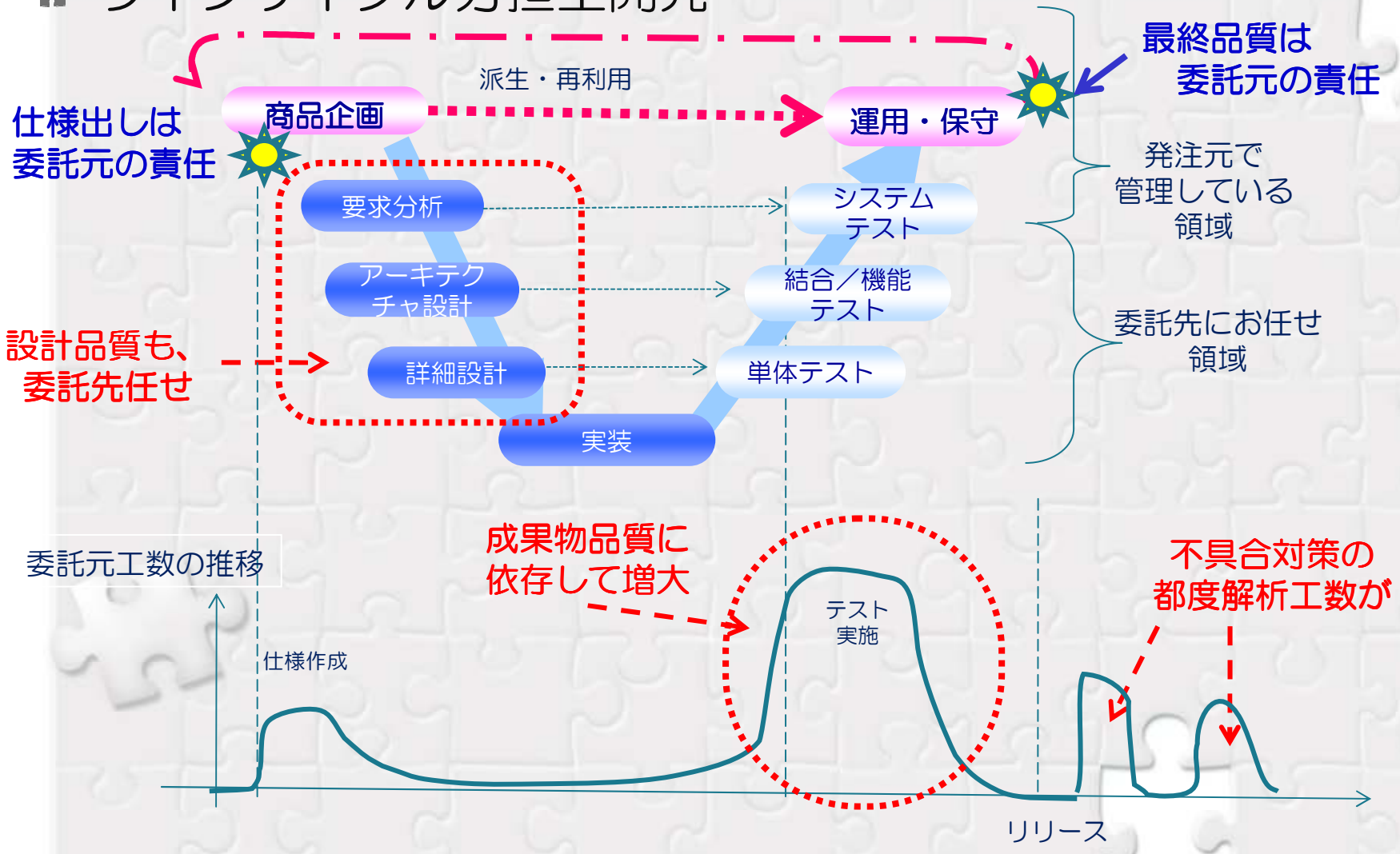
- 開発は、委託先の信頼任せ
  - コストの妥当性は・・・
    - 開発期間、工数の妥当性が判断ができない
  - 将来的な保守も、委託先頼りに・・・
    - 出荷後の問題も、解析できない
    - 修正・変更があっても、手を入れられない

委託先の依存度合が
増すばかりに

→ 成果物は暗黙状態
ソフトウェア資産価値が
上げられない

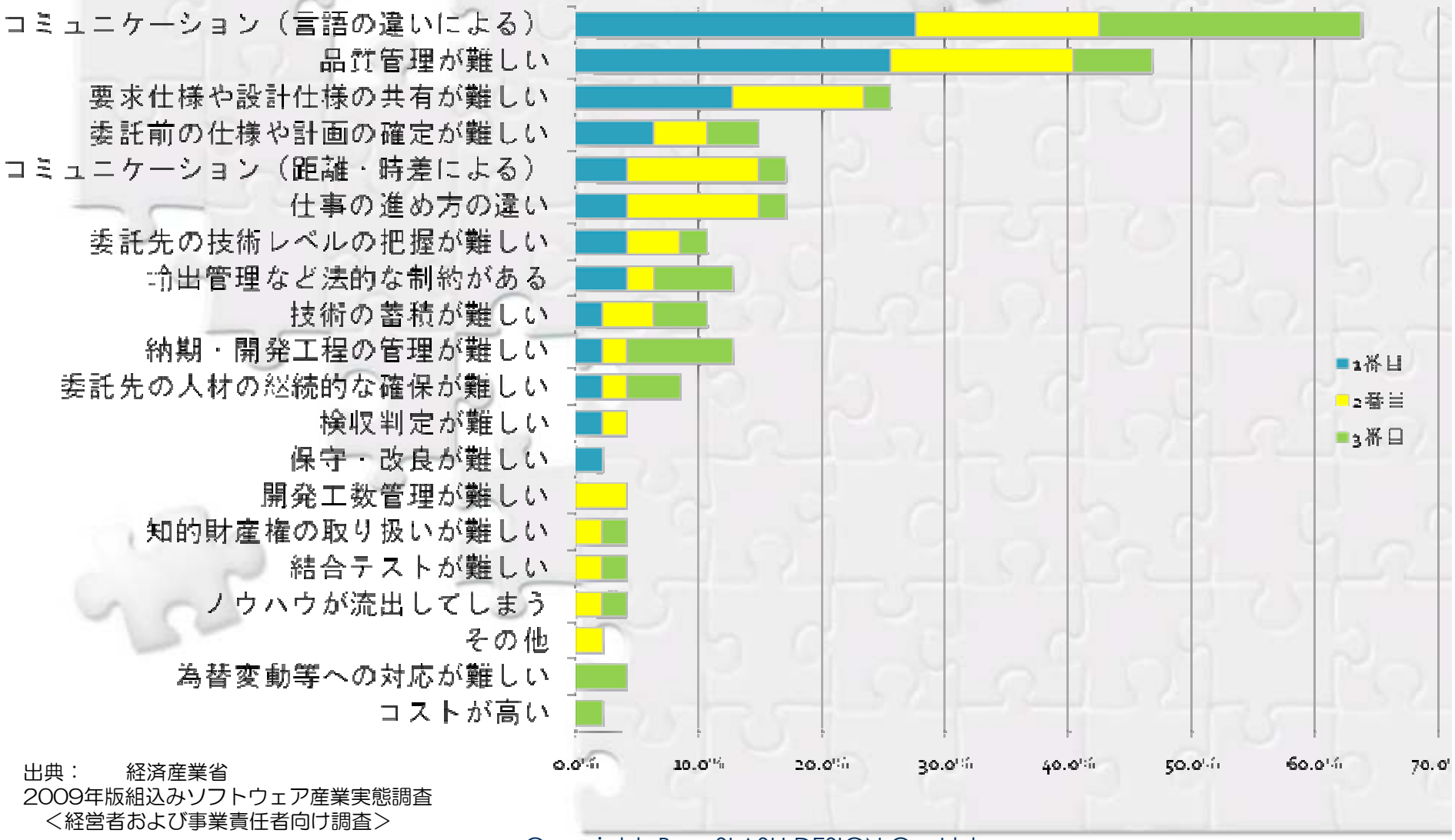
オフショア開発の例

- ライフサイクル分担型開発



オフショア開発の問題点

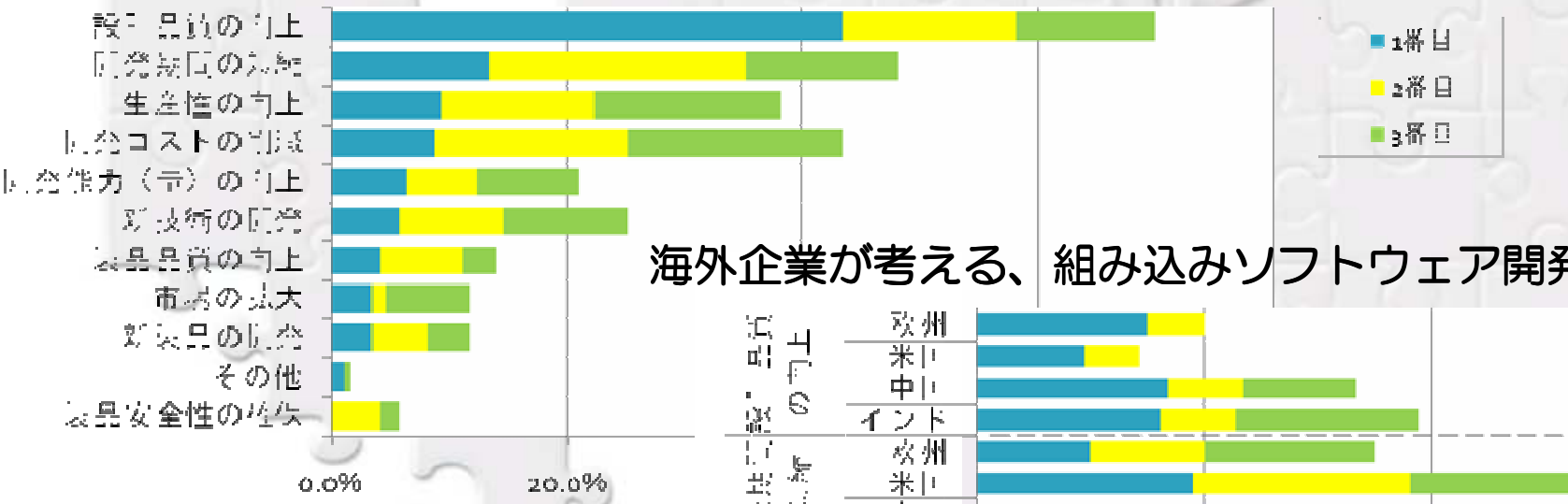
外部委託の課題　（委託先：海外企業）



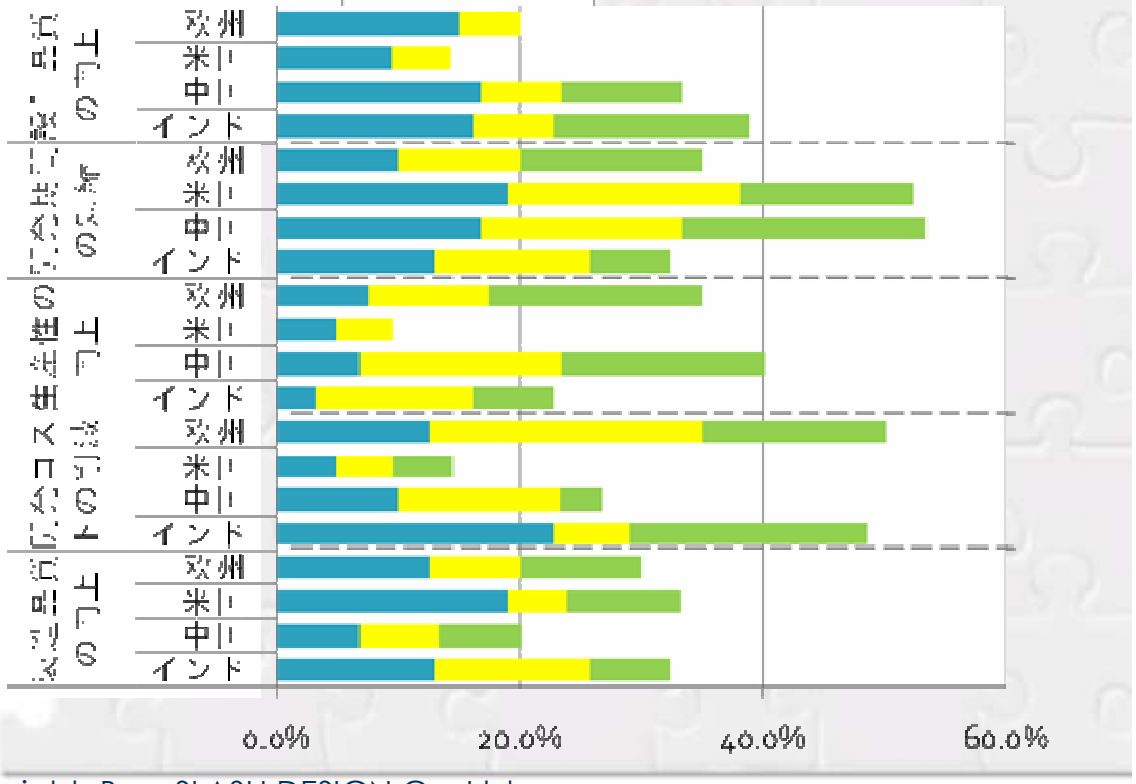
出典：　経済産業省
2009年版組込みソフトウェア産業実態調査
＜経営者および事業責任者向け調査＞

日本の課題意識と、他国の課題意識

日本企業が考える、組み込みソフトウェア開発の課題



海外企業が考える、組み込みソフトウェア開発の課題



出典：　経済産業省
2009年版組込みソフトウェア産業実態調査
＜経営者および事業責任者向け調査＞
＜海外調査＞

※ 欧州には、ロシアも含まれる
※ 韓国・台湾・東南アジア　の回答は、日本企業の傾向に酷似しているため、グラフには載せていない

本事例：委託側の希望

- ソフトウェアの中身が把握したい
  - 設計ドキュメントがない
    - → 内部品質の確認ができない
      バグ修正、仕様変更にかかる工数が予想できない
  - 言葉の壁があり状況把握が困難
    - → 要求が正しく伝わっているか不安がある
      遅延発生時、何が問題で、なぜ困っているのかわからない

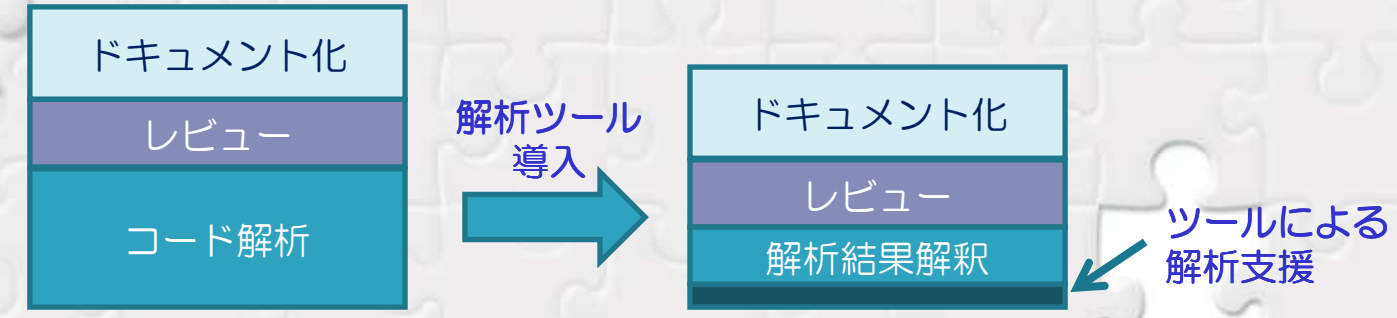
- 解析に時間をかけたくない
  - 開発済みコードのテスト（結合~システム）準備にも工数が必要
    - → 新たに解析工数を割くことが難しい
  - アーキテクチャを知らずにBlack Boxテストには不安がある
    - → 変更確認時も、某大なテストケースを実施
      計画的なテストの実行にも問題が…

設計を把握し、委託先任せからの脱出を！

ご提案：

ソフトウェアの静的構造を可視化
⇒　委託元で検収作業を計画的に進める

- 施策１
  - アーキテクチャの把握できるドキュメントを作成する
    ⇒　納品物の可視化により、*検収量を見える化*する
    内部構造を把握して、*テスト対象範囲を見える化*する
- 施策２
  - 静的解析ツール導入により、可視化にかかる工数削減
    ⇒　ツールで時間のかかる作業を削減　・・・　解析漏れの防止



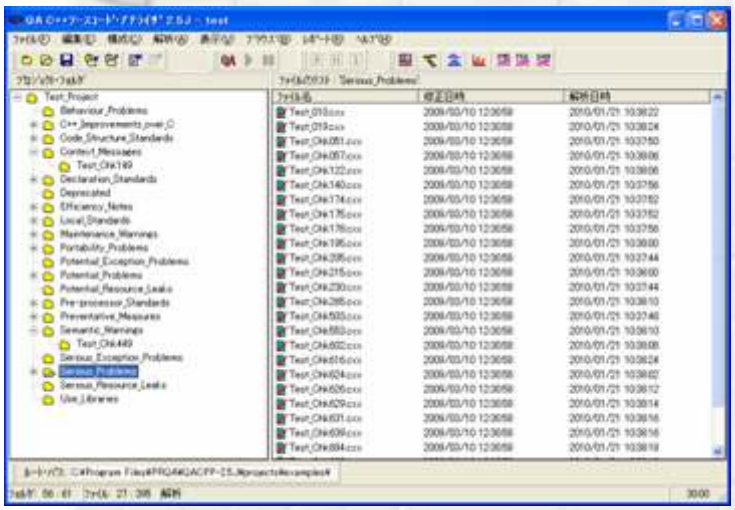
Software静的解析ツール

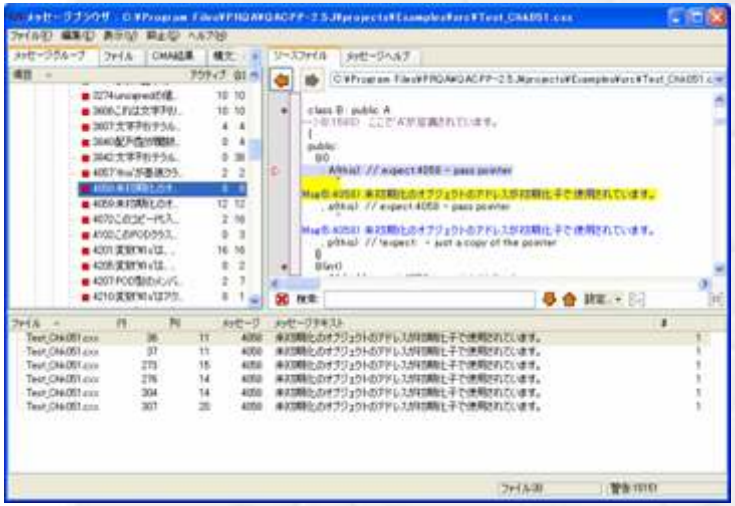
- 静的解析ツール
  - ソースコードを起点として、ソフトウェアを動かさずに解析し、その解析結果を可視化する。
- 機能
  - コードのチェック
    - 構文エラー、記述ミスのチェック
    - バグ要因になりやすい記述の警告
  - メトリクス（評価指標）の算出
    - 信頼性、保守性、移植性　etc. を検討する目安
    - 過去データから、閾値を割り出す
  - モジュール内部構造や依存関係を図・表化
    - コードから、プログラム構造をリバースする
      - アルゴリズムをモデル図に可視化
      - 各種の関連性をモデル図に可視化

など…
いろいろなツールがある

BACKSLASH DESIGN

C++言語静的解析ツール QA C++



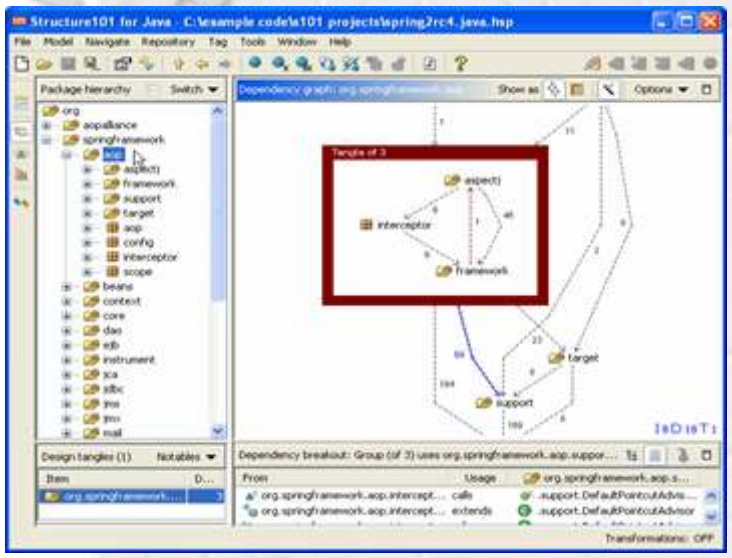


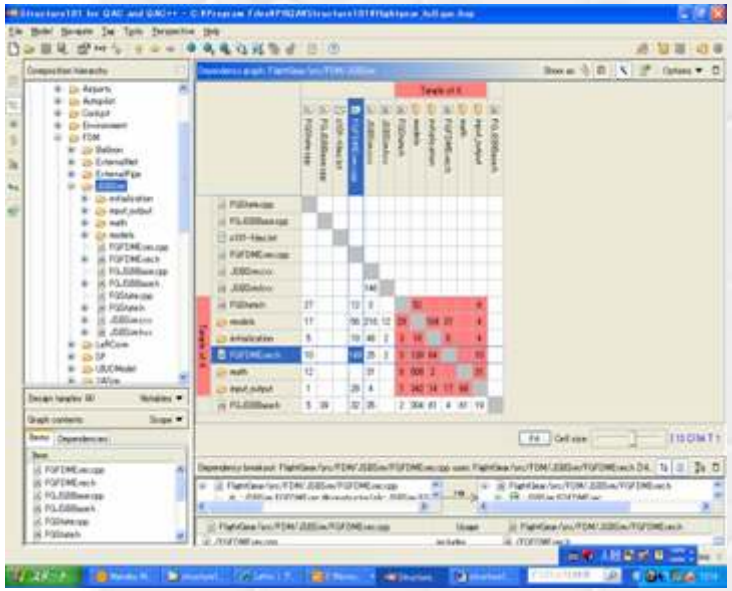
【特長】

- C++言語ソースコード静的テスト
  - -ISO/IEC C++（14882:2003）標準規格準拠
  - -約1300個のチェック機能により問題点を抽出
  - -強力なコードレイアウトチェック機能
- 業界標準のソフトウェア・メトリクスによる品質定量化
- コーリングツリー、クラス継承図等によるビジュアル化
- メッセージブラウザにより警告箇所を容易に参照可能
- オンラインヘルプによる警告内容の詳細説明
- 開発環境に容易にアドオン可能
- 容易なグラフィカルユーザインターフェース
- コマンドラインによる解析処理の自動実行が可能
- 解析後の処理に命名規則チェックなどの追加が可能
- Microsoft Visual C++6.0/.NET2003/2005/2008に対応

PR:QA 株式会社 東陽テクニカ

アーキテクチャ分析支援ツール Structure101





【特長】

- アーキテクチャダイヤグラムを用いた設計要素のグループ化と設計構造の可視化
- ＤＳＭを用いた設計要素のグループ化と設計構造の可視化
- 依存関係図を用いた設計要素のグループ化と設計構造の可視化
- 相互依存と経路の数、依存の数を用いて構造上の複雑さを検出（XS値）
- 一定期間毎にデータをリポジトリに保存し、全体のコードサイズやXS値の遷移を表示
- 予め定義したアーキテクチャの設計図と実コードとの整合性を確認

解析手順の紹介

- ソースコードから、主要アーキテクチャが把握できるような図面を作成

  ⇒ リバースモデル

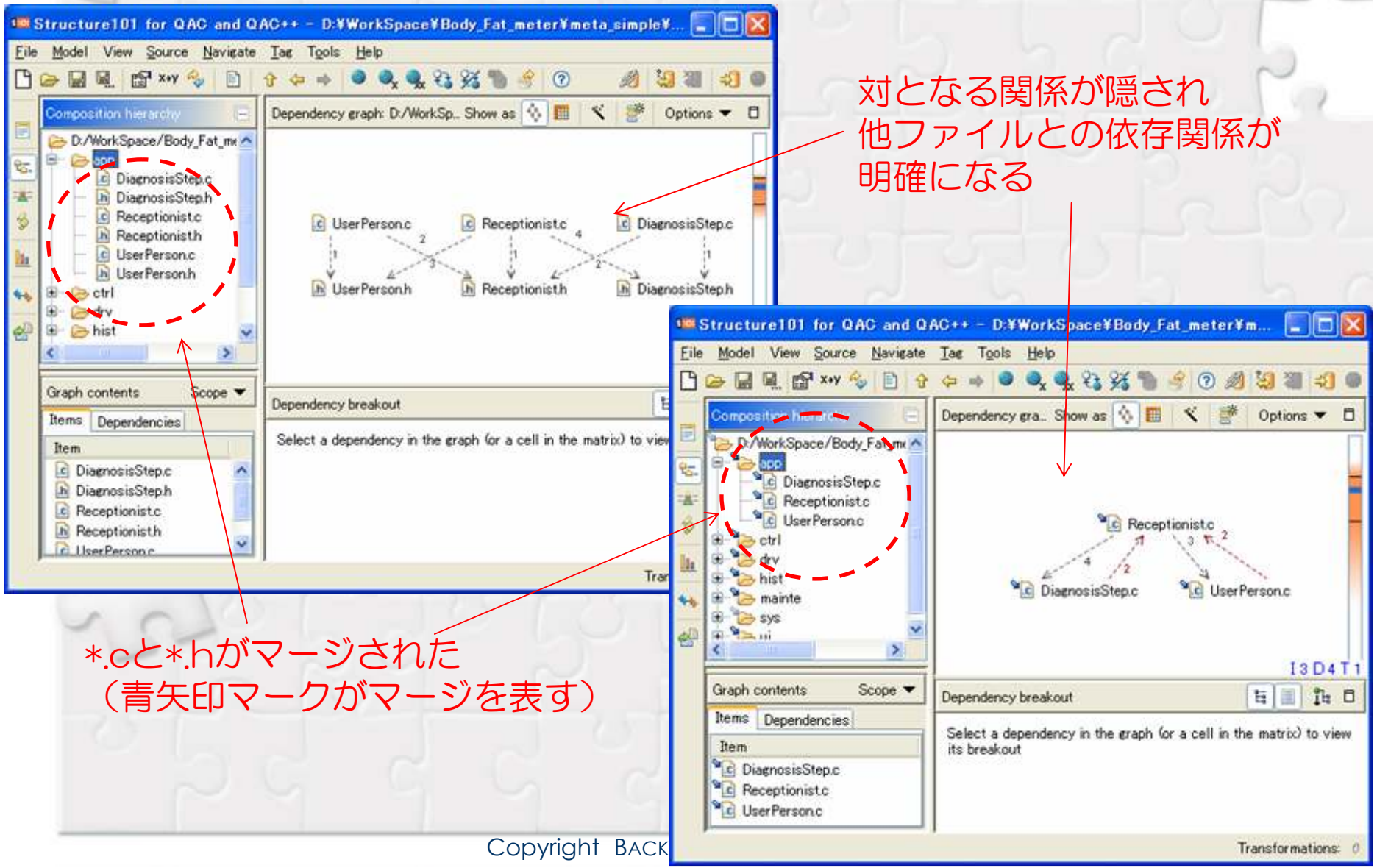
本件の主な解析ステップ

- ステップ1　実装構造の把握
- ステップ2　依存関係を分析
- ステップ3　設計意図に基づく再構築

解析第１ステップ：実装構造の把握

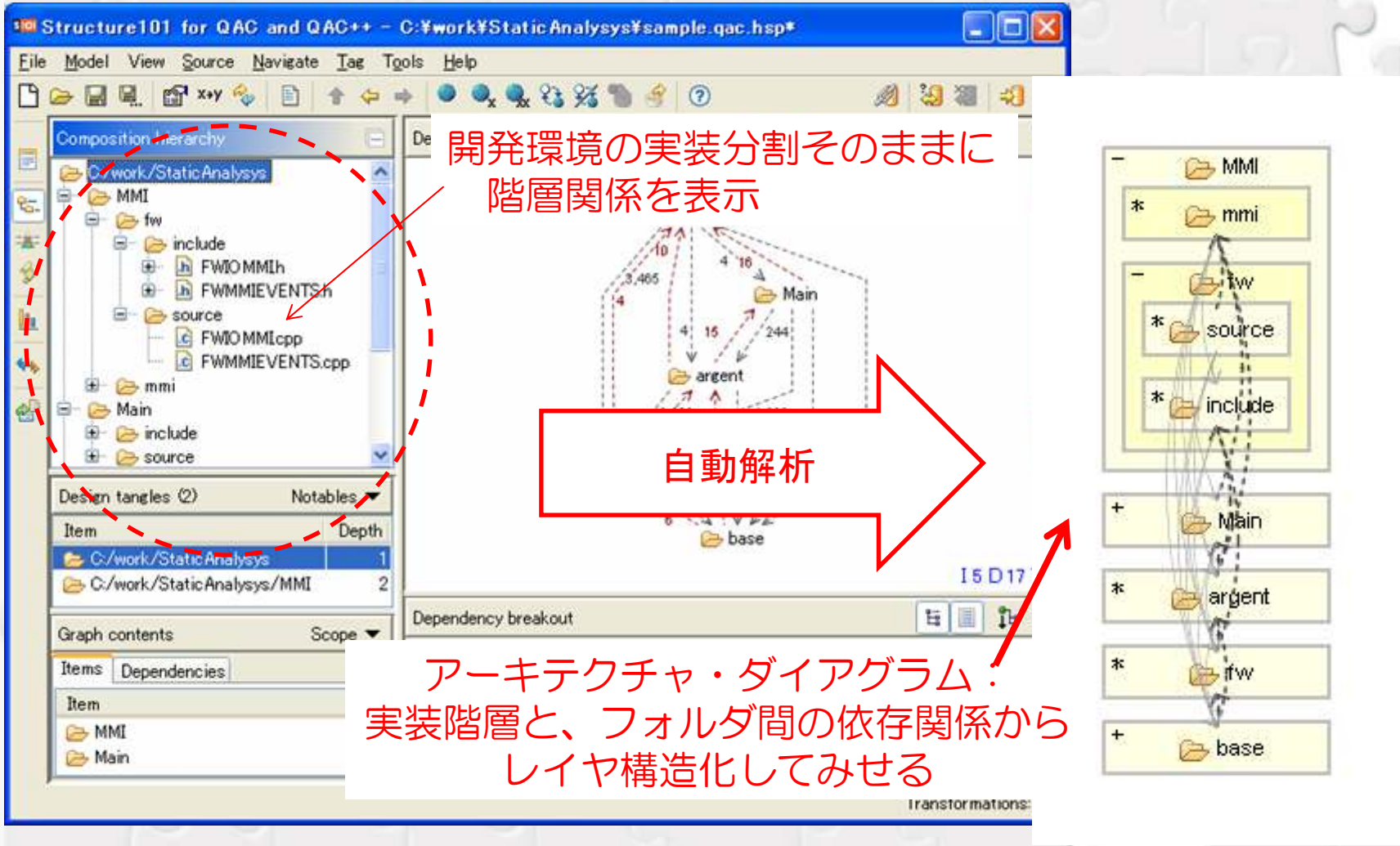
- プログラムコードの整理
  - ヘッダ・ソースファイルのパッケージ化
    - *.hは、*.cppの外部I/F窓口
    - 他ファイルとの関連を見るとき
      対となる*.hと*.cppはひと固まりとして見たい
      注：*.cppの外部公開宣言が同名*.hに書かれていることが前提
- 依存関係から、ありのままの構造を把握
  - 現在の依存関係から、レイヤ構造を推察
    （ツールによる自動解析）
- プログラムコードの管理単位を分析
  - 実装時の分割
    - 開発時、フォルダ分割した何らかの意図がある
    - フォルダ階層、ファイル名から当初の意図を推察する

ソース・ヘッダーのパッケージ化
*.hのマージ例



ありのまま(実装)構造の把握

- フォルダ・ファイルの所在



管理単位から、設計時の意図推察

- フォルダ ＝ 開発時の管理単位
  - ソフトウェア部品の入れ物
    - 部品（ファイル）をグループ分けして格納しているもの
- ファイル＝　システムを構成する部品(パーツ)
  - ファイル分割の軌跡
    - ファイル名、コメント

⇨ 部品の特徴・概略を示す

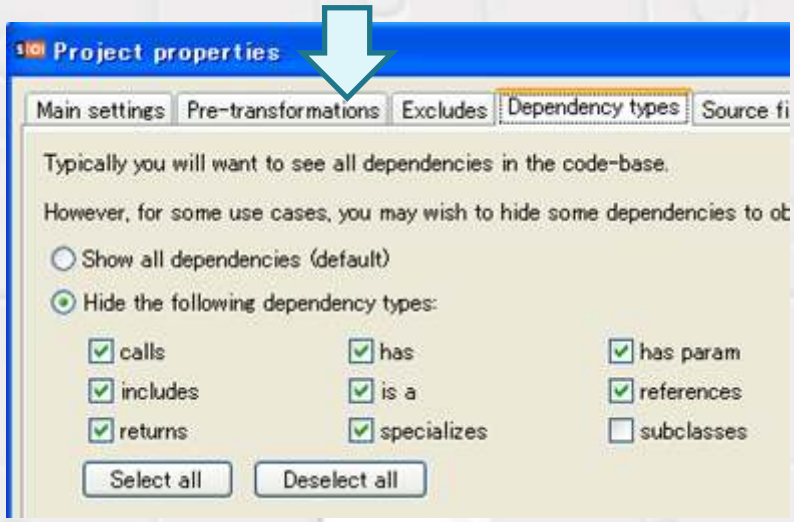
- 部品の分割・分類の方針
  - 動作環境に依存した分割、作業者に依存した分割
  - 既存の再利用部分
  - 機能・制御対象物など、役割による分類　　etc.

15

解析第2ステップ：依存関係の分析

- 依存関係から設計構造を推察するための材料を集める
  - クラス継承関係からの分析
    - クラス群から、役割によるグループ分けを作る
    - 基底クラスと、サブプラスの実装状況を調査する
  - Has関係からの分析
    - 集約関係を持つクラスを探す
  - 呼び出し関係からの分析
    - クラス間の関連性を調査する
      - 呼ぶ側と呼ばれる側で
        上下関係となるクラスを探す
      - メッセージの受け渡しで
        横並び関係となるクラスを探す

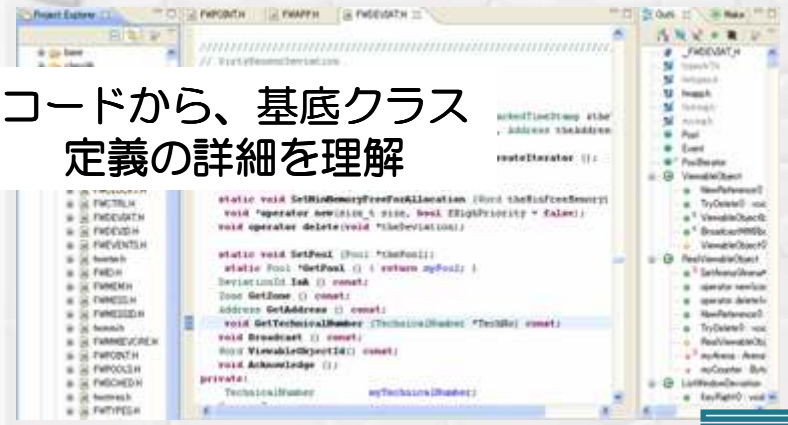
ツール設定により依存関係種別を
絞って、効率よく調査を！



例：クラス継承関係の分析

クラス継承関係から
同種の役割を担うファイル
グループをまとめてゆく

ツールを活用して、分析作業を効率化



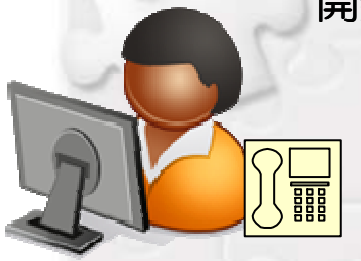
コードから、基底クラス
定義の詳細を理解

クラスグループの固まり
実装時の分散具合を調査

Structure101 で
クラス継承に的を絞った
現状を表示させる

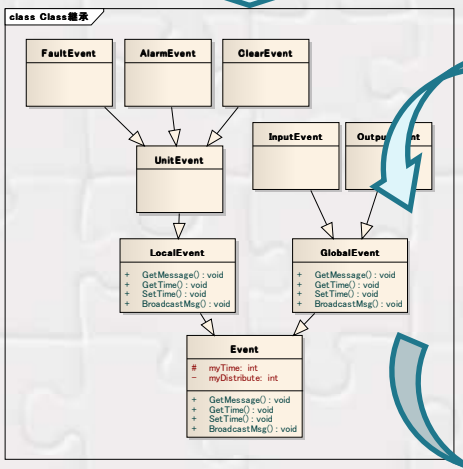
調査・分析結果をクラス図で整理

開発元
担当者



具体的な不明点を
開発元に質問

ポイントを回答



SW仕様書中から、
そのクラスの役割を
考えてみる

解読事項を
記録

ステップ３：設計意図に基づく再構築

- 分析結果を組み立てる
  - クラス継承・集約、呼び出し関係から分析した結果を分類する
    - クラス間の階層構造
    - 横並びの役割分担関係
    - 共通処理部
  - アーキテクチャを推察
    - Class名、関数名から各モジュール役割を解読
    - 他のモジュールとの関係から、役割分担を解読
      - コードの詳細解析、動的な追跡　etc.
    - 解読した結果を、アーキテクチャ図面に起こす

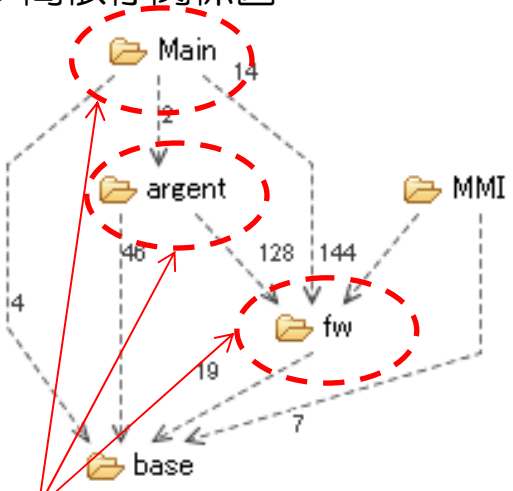
第２ステップの分析結果

第３ステップで新たに考察

実装構造から意図を推察

- 現フォルダ・ファイル、各々の役割を分析
- 設計文書があれば付き合わせ、
  設計構造の崩れが生じていないか確認する

フォルダ間依存関係図



フォルダごと、
(ファイルごと)の
役割を推察

設計図
・機能構造図
・Class図　　etc.

設計者

設計資料

設計段階で決められた
アーキテクチャ
（ソフトウェア構造）

乖離が生じる

実装者

プログラムコード

コーディングされた
ソフトウェア構造

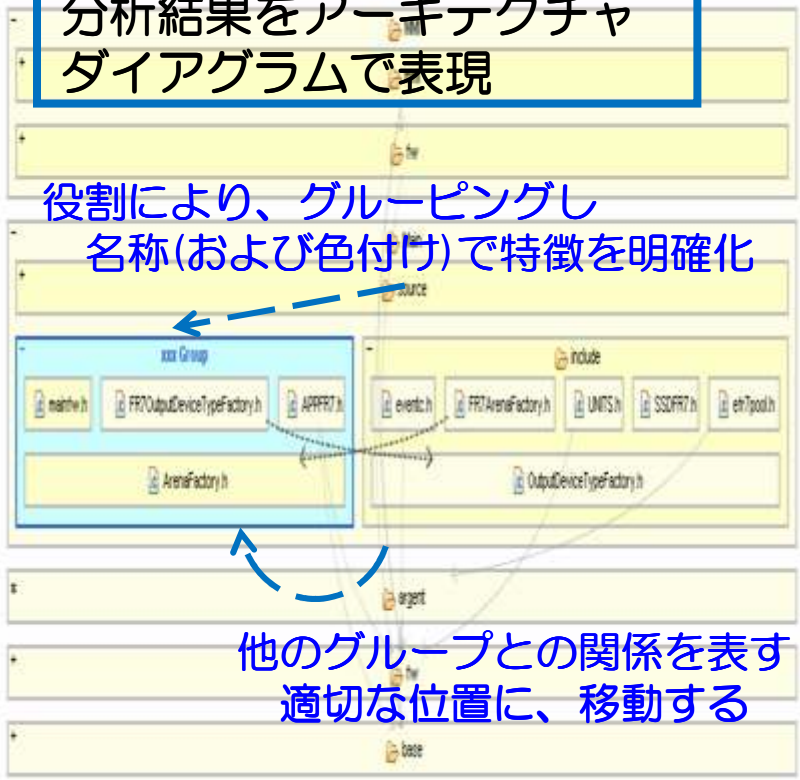
BACKSLASH DESIGN

分析結果を組み立てる

- アーキテクチャダイアグラムを使って、
  発掘した設計意図に応じた構造を表現

自動で作成した
アーキテクチャダイアグラム

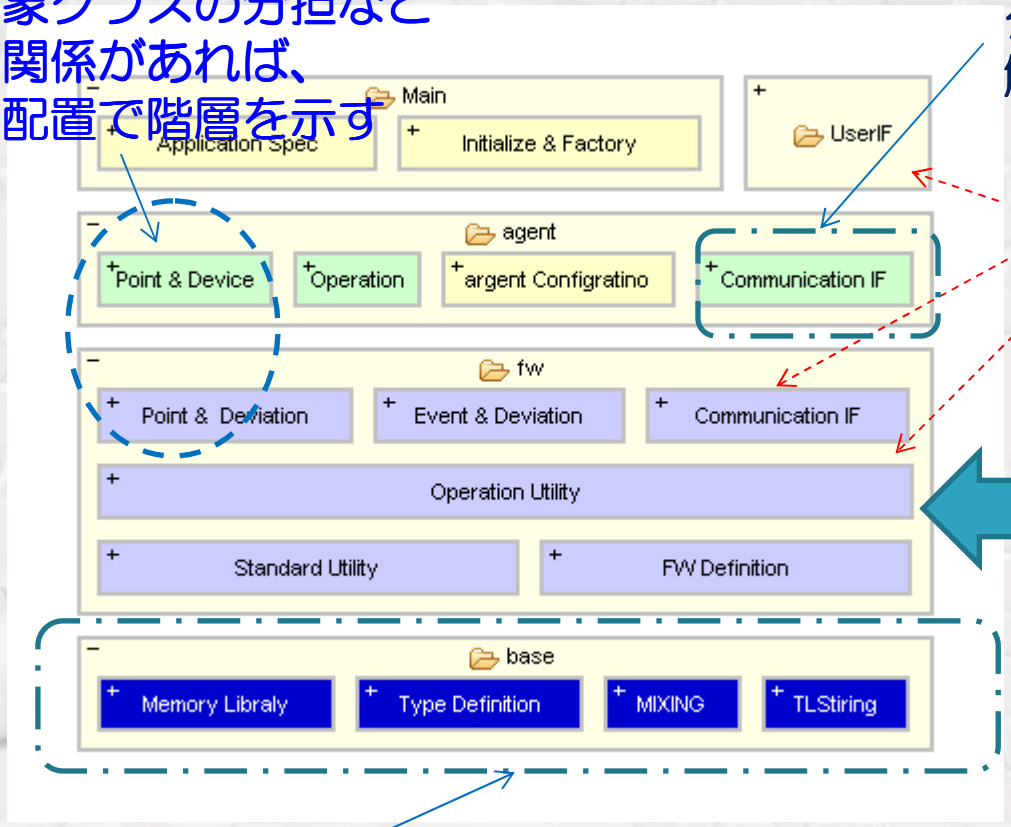
分析結果をアーキテクチャ
ダイアグラムで表現



分析結果を組み立てる(続き)

- 設計意図を反映したアーキテクチャダイアグラム

抽象・具象クラスの分担など
階層関係があれば、
セルの配置で階層を示す



グルーピングしたセルには
解読した役割名を書く

同種の役割を担う
ファイルグループは同じセル
にラッピング

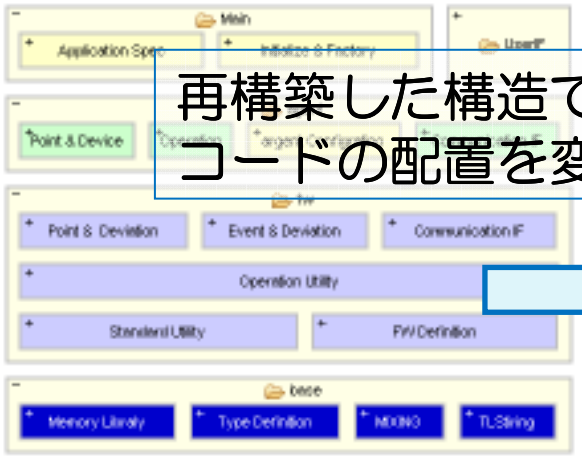
同階層（横並び）の
クラス関係は
役割分担を考察して配置

別途クラス図等のモデル等を作成する

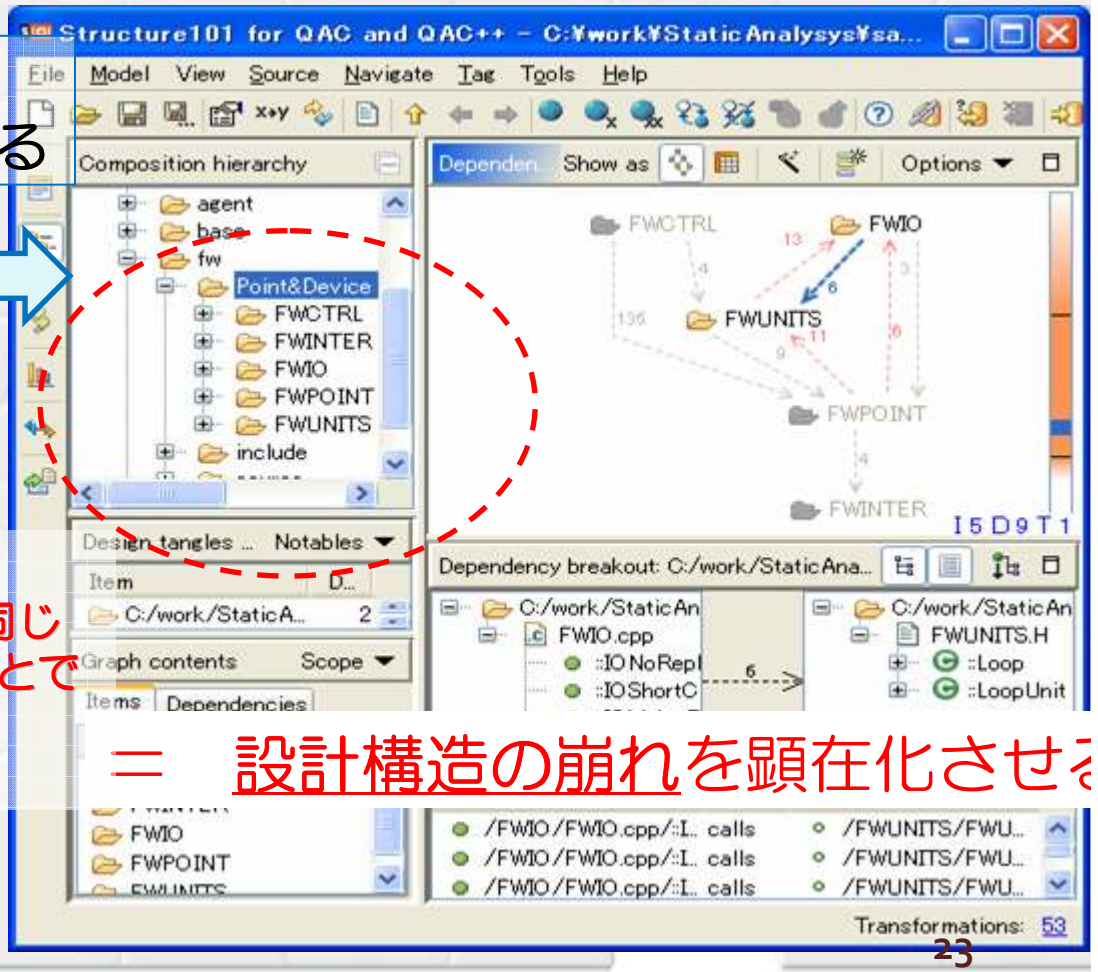
共通処理部を
下位層へ・・・

再構成した構造で、依存関係確認

- 設計意図に合わせたコンポーネントで、依存関係見る



再構築した構造で、
コードの配置を変更する

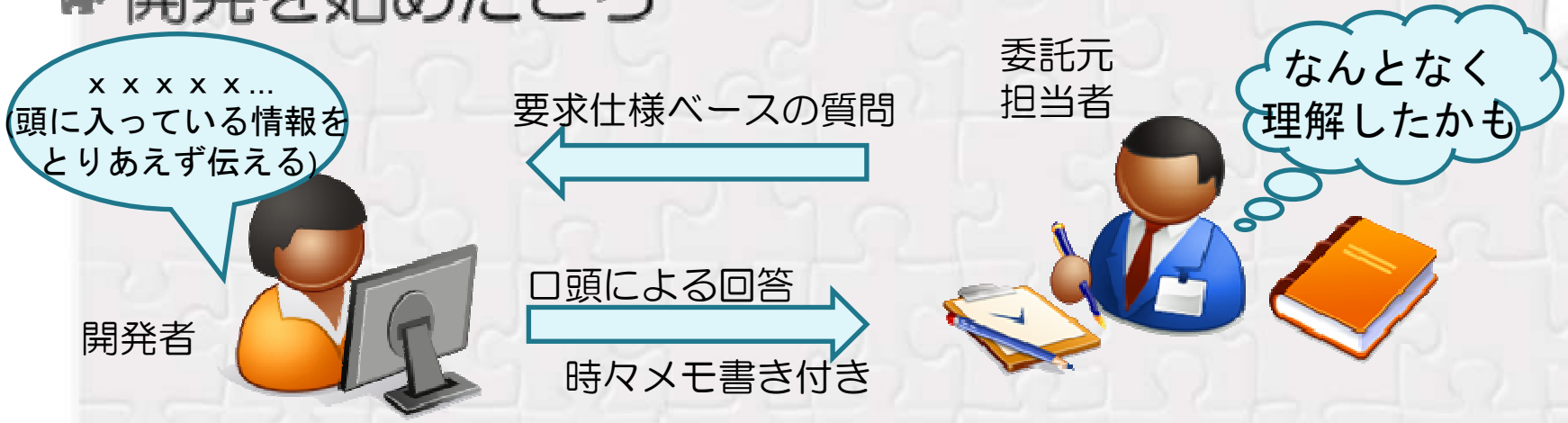


Transformation機能により
アーキテクチャダイアグラムと同じ
フォルダ分割を仮想的につくることで
設計意図による依存関係を
可視化することができる

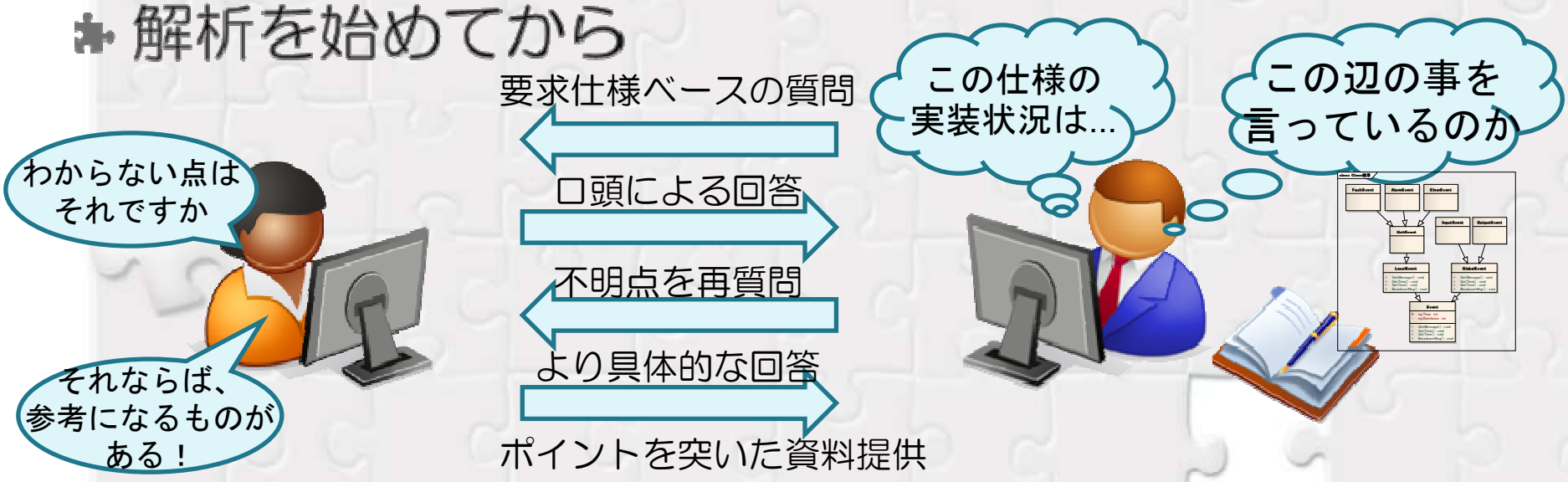
＝　設計構造の崩れを顕在化させる

開発元との対応の変化

開発を始めたころ

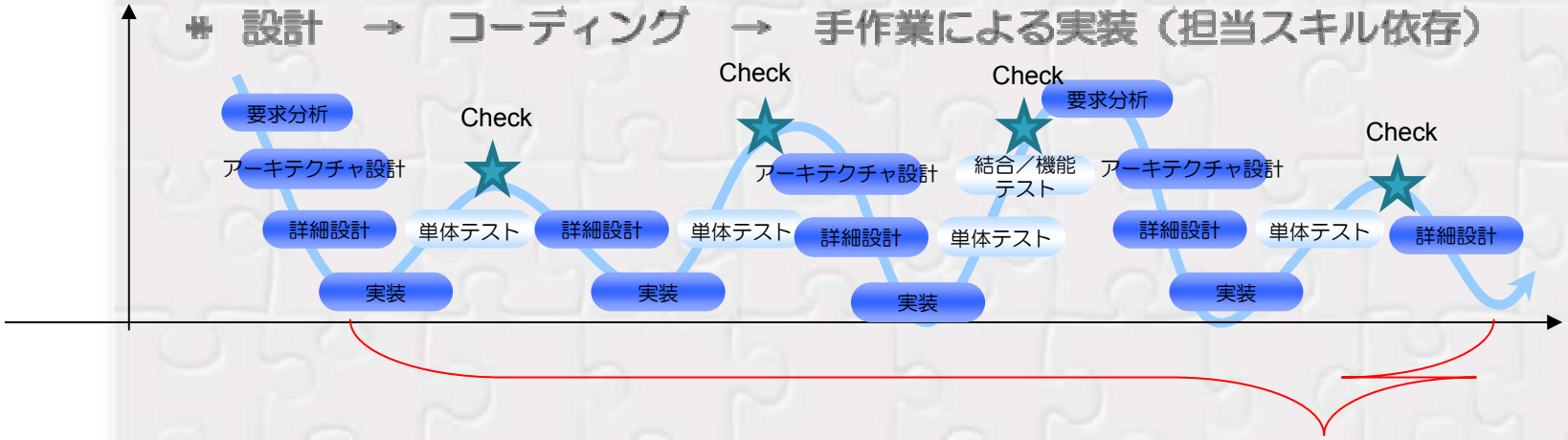


解析を始めてから



リバースモデル活用１：設計品質の確認

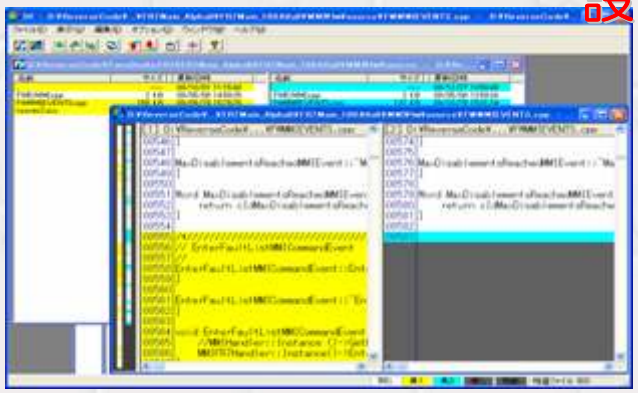
- 開発途中、できた部分から仮リリース
  - 分析→設計→実装→テスト・・・　繰り返し
  - 設計　→　コーディング　→　手作業による実装（担当スキル依存）



全体を通して、アーキテクチャ設計思想は保たれているのか確認できていますか？

- 品質確保は、どうしていますか？

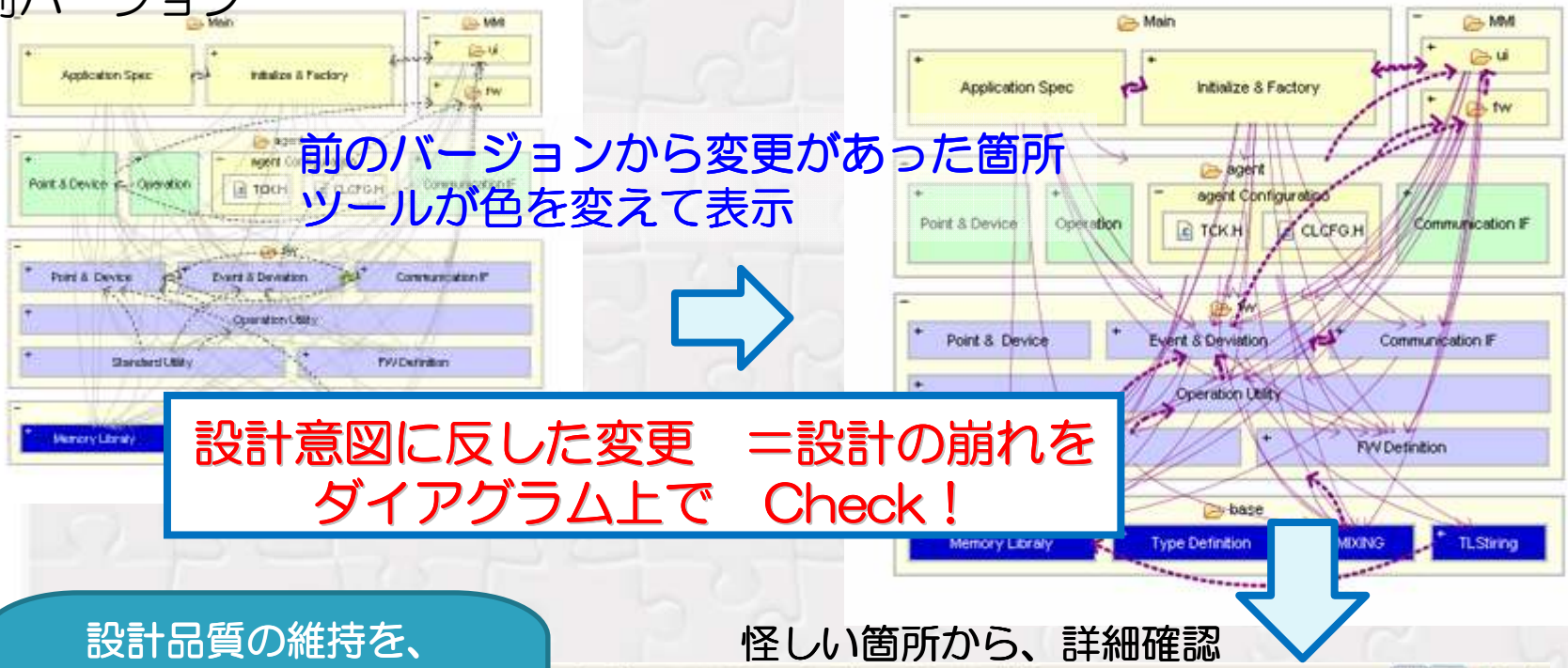
~~CodeのDiffで確認~~



設計品質は、モデルでCheck！

前バージョン

新バージョン



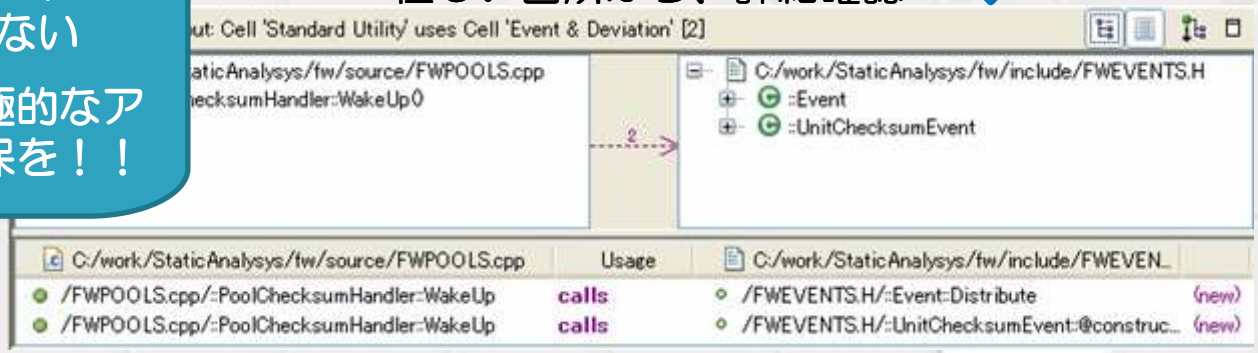
前のバージョンから変更があった箇所
ツールが色を変えて表示

設計意図に反した変更　＝設計の崩れを
ダイアグラム上で　Check！

設計品質の維持を、
開発元任せにしない

委託元で管理、積極的なアプローチで品質確保を！！

怪しい箇所から、詳細確認

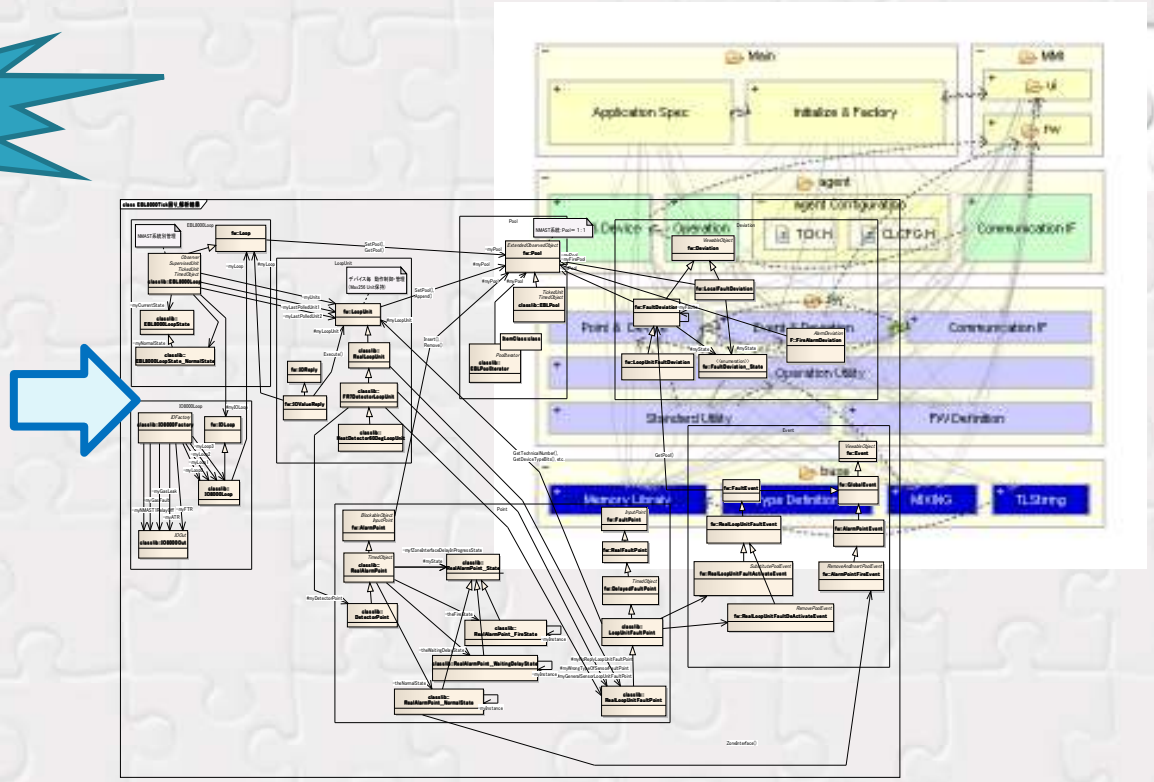


リバースモデル活用２：計画的なテストの実施

不具合が
発生

委託元
担当者

開発者に原因・修正
箇所を問い合わせる

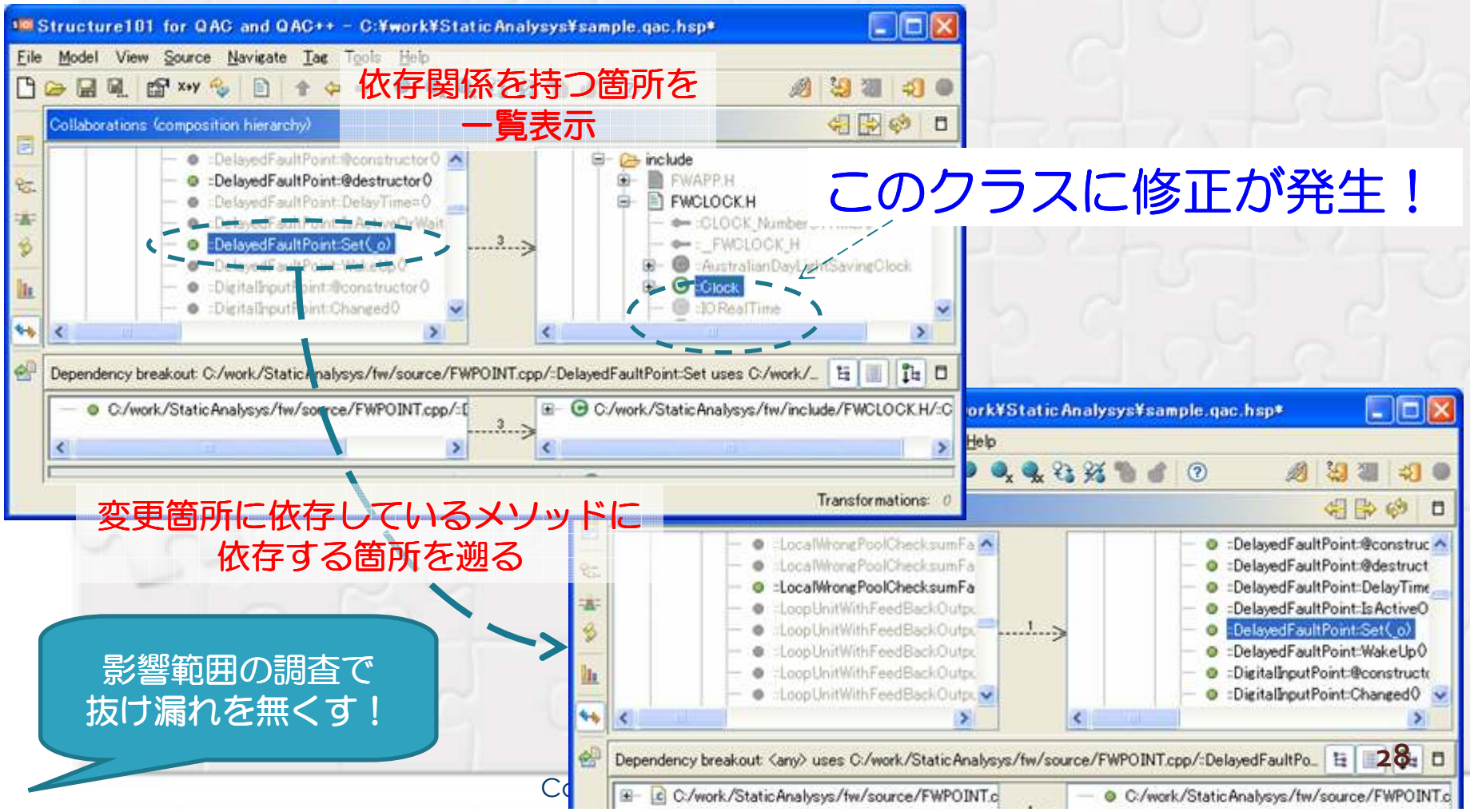


設計情報とリバースモデルで、
変更箇所を理解する

変更箇所の影響範囲を調べて
テストの準備を！！

影響範囲は、ツールで調べる

- 変更予定箇所が判明　→
  影響範囲を、ツール上で追いかける



オフショア開発に“壁”はつきもの

- コミュニケーションが取りにくい
  - 仕様のとり違い、変更の説明が困難
  - 品質要求に対する、認識の違い
- 品質確保は、最後の最後になる
  - ブラックボックスによる、システムテスト
  - 委託先の提出する、動作確認書
- 開発は、委託先の信頼任せ
  - コストの妥当性は・・
    - 開発期間、工数の妥当性が判断ができない
  - 将来的な保守も、委託先頼りに・・・
    - 出荷後の問題も、解析できない
    - 修正・変更があっても、手を入れられない

開発元の出方を待って対応していては、“言葉の壁”は越えられません

待ち受け体制から、
積極的に働きかける体制に転換

本事例で紹介した静的解析による可視化は、
積極的なアプローチをするための、一つの道具になります。

BACKSLASH DESIGN

ご清聴ありがとうございました。

JaSST’10　東陽テクニカ・テクニカルセッション

アーキテクチャ分析ツールStructure101による
オフショアプログラム解析事例