C/C++ ソースプログラムのアーキテクチャを見える化

株式会社東陽テクニカ
ソフトウェア・ソリューション

Static Analysis Solutions

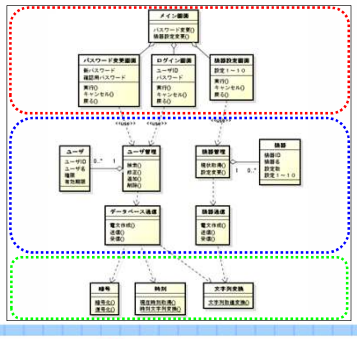
東陽テクニカ PRQA

目次

- アーキテクチャの崩れとテスト
- アーキテクチャの崩れに気づくには
- Structure101の紹介
- 設計図と実コードの比較

アーキテクチャとは(1/2)

ソフトウェアの全体構成/枠組み



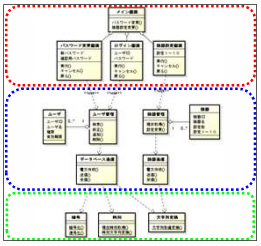
GUI(ユーザ I/F)

通信

汎用計算

アーキテクチャとは(2/2)

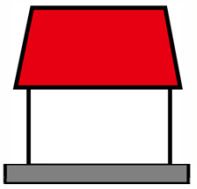
家に例えると・・・



屋根

壁

土台



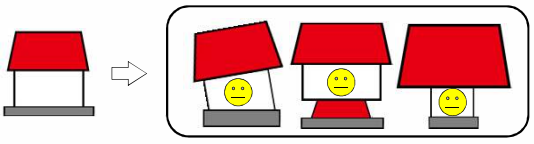
目次

- アーキテクチャの崩れとテスト
- アーキテクチャの崩れに気づくには
- Structure101の紹介
- 設計図と実コードの比較

アーキテクチャ崩れの例(1/3)

アーキテクチャが崩れると、テストが大変になる

- 家の場合



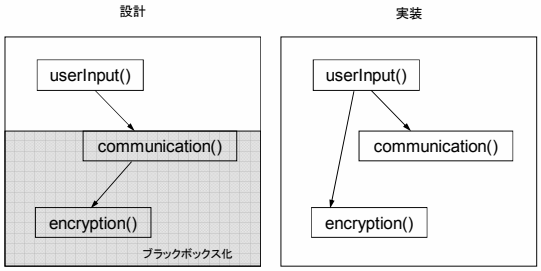
家の中にいる人は、崩れに気づきにくい

=

ソフトウェア開発者は**家の中にいる**状態

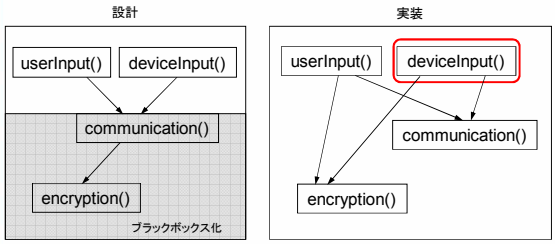
アーキテクチャ崩れの例(2/3)

◆ ソフトウェアの場合



アーキテクチャ崩れの例(3/3)

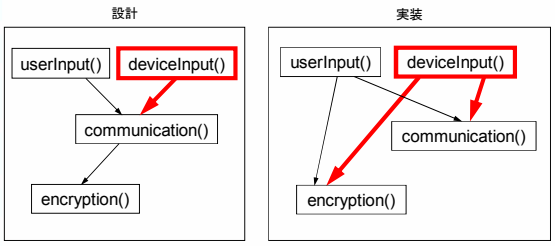
◆ 機能拡張時



このような実装を繰り返すことで
アーキテクチャはさらに崩れる

機能拡張時のテスト範囲

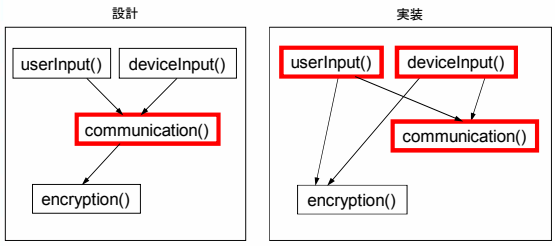
◆ deviceInput () 追加時



deviceInput() → encryption()
I/F確認の分、テスト範囲が大きい

仕様変更の範囲

◆ 通信フォーマット仕様変更時



影響範囲が大きい

アーキテクチャの崩れとテスト

アーキテクチャが崩れていると、影響範囲が大きくなりすぎる

⇩

テストが大変になる

◆ 本来、必要な箇所以上のテストが必要になる

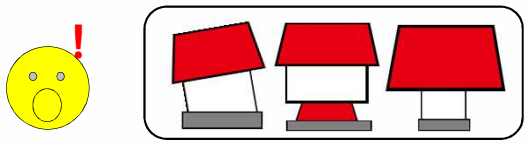
◆ 無関係と思っていた箇所に影響があり、
デグレードが発生する

◆ 問題が発生した場合、
問題の絞込みに時間がかかる

目次

崩れに気づくには

◆ 家を外から見る



外から見ると崩れていることが分かる

||

ソフトウェア全体として
設計どおりに実装されているかを見る

設計と実装の比較手段(1/4)

◆ ソースコードとディレクトリ構造

設計図

関連を示すのは
ファイル名頼り？

実装状態

ソースコードとディレクトリ構造から
設計どおりかを判断するのは困難

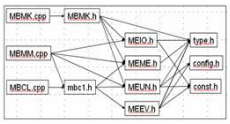
設計と実装の比較手段(2/4)

◆ インクルード関係や呼び出し関係

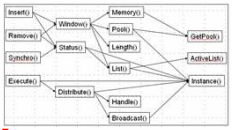
設計図

設計図の
どこに対応？

インクルード関係



呼び出し関係



情報としては必要だが
ファイルのインクルード関係や
関数の呼び出し関係だけでは不十分

設計と実装の比較手段(3/4)

設計図

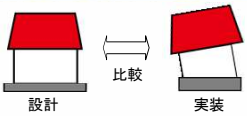
GUI(ユーザ I/F)

通信

汎用計算

「グループ化」と「グループの上下関係」で
設計図に近づけた形で可視化

設計と実装の比較手段(4/4)



ソースコードならStructure101を用いると
設計に近づけた形で実装を表示することができる

例えば、次のような問題点が分かる

- ◆ 下位層から上位層への依存がある
- ◆ 様々な箇所と依存関係があり複雑すぎる
- ◆ 階層を跨ぐ依存関係がある

目次

Structure101とは

静的解析ツールQA C/QA C++に
アドオンして使用するアーキテクチャ分析支援ツール

【特長】
- ◆ コードの構成要素間の依存関係を表示
- ◆ コードの構成要素の階層構造を可視化
- ◆ 構造上、好ましくない依存関係(循環依存など)を検出
- ◆ コードの階層構造の設計図(アーキテクチャ図)と実コードの比較
- ◆ コードの改版の影響範囲を表示
- ◆ コードの改版で追加された要素や依存関係を強調表示
- ◆ Webブラウザから参照可能

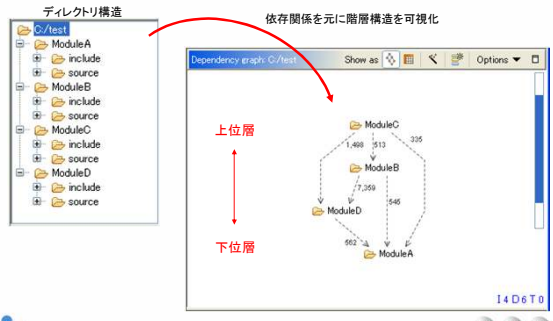
依存関係とは

コードのある要素が別の要素に影響を与えるとき、依存関係がある
フォルダ、ファイル、グローバル変数、関数、クラス・・・

・includes	ファイルAがファイルBをインクルードする
・calls	関数Aが関数Bを呼び出す
・references	関数Aがクラスまたはグローバル変数Bを使用する
・has	クラスAがクラスBをメンバに持つ
・has param	関数AがクラスBをパラメータに持つ
・returns	関数Aがクラスを返す
・specializes	関数Aがテンプレートを特殊化する
・subclasses	クラスAがクラスBを継承する

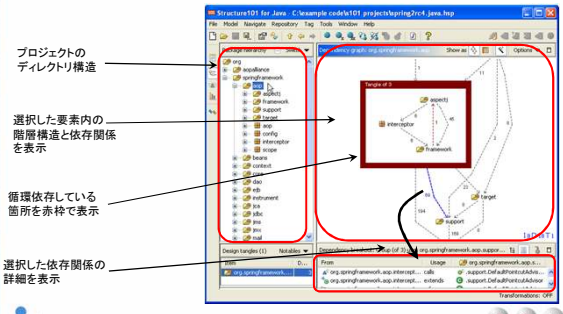
要素Aは要素Bに依存する　A → B

階層構造を可視化(1/2)



階層構造を可視化(2/2)

階層構造をグラフで表示する。矢印で依存の方向、数字で依存の数を表す。



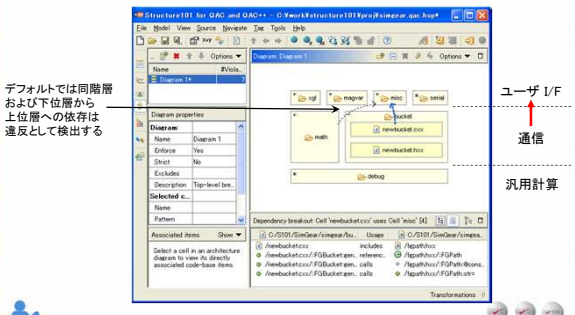
設計図の作成(1/3)

既存のコードから設計図を自動生成することが可能

- ◆ 設計図にはフォルダやファイルがセルとしてマッピング
- ◆ セルの追加、削除、移動、グルーピングなどが可能

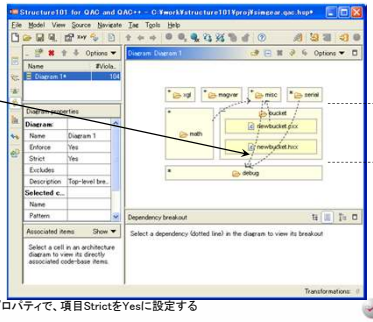
設計図の作成(2/3)

下位層から上位層への依存関係を検出



設計図の作成(3/3)

階層を跨ぐ依存関係を検出



上位層から下位層への依存であっても階層を跨ぐ依存を違反として検出する(※)

また、設計図上の任意の要素間の依存に対して、違反とする/しないを設定可能

ユーザ I/F

通信

汎用計算

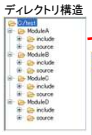
※設計図のプロパティで、項目StrictをYesに設定する

目次

- アーキテクチャの崩れとテスト
- アーキテクチャの崩れに気づくには
- Structure101の紹介
- 設計図と実コードの比較

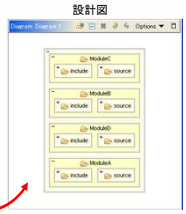
比較のための準備(1/2)

ディレクトリ構造



階層構造

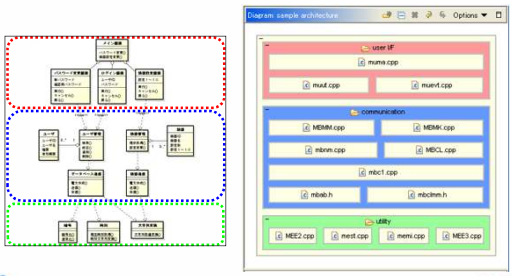
設計図



- 階層構造はツールが自動的に判断
- ツールがアーキテクチャの設計意図を汲んでくれる訳ではない

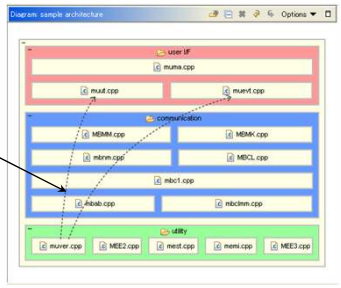
比較のための準備(2/2)

少なくとも初回は
設計意図を知っている「アーキテクチャ設計者」が設計図を調整



問題例1(1/2)

- 下位層から上位層への依存がある



汎用計算(utility)からユーザI/F(GUI)への依存関係がある

この汎用計算のプログラムを別のアプリケーションで使用する場合、GUI画面まで必要になってしまう

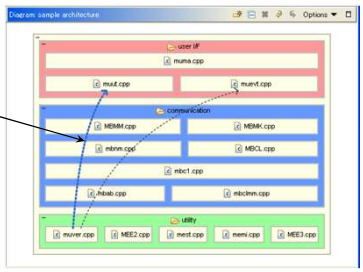
→ 再利用性の低下

問題例1(2/2)

実装の誤り

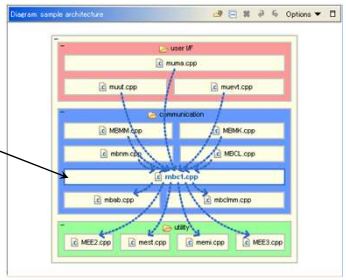
GUIのコード内に本来は汎用計算にあるべき関数が含まれていて下位層から呼び出されている

対象の関数を分離し汎用計算へ



問題例2(1/2)

◆ 様々な箇所と依存関係があり複雑すぎる



下位層から上位層への依存はないがこのソースは、下位の様々なソースから影響を受け、上位の様々なソースに影響を与える

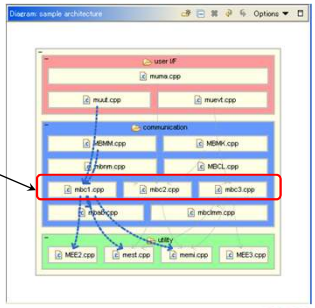
→ コードが肥大化し処理が複雑になっている可能性

問題例2(2/2)

ファイルの分割

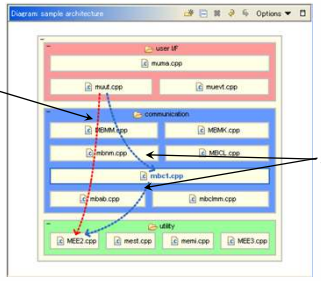
コードを分割することで依存関係を簡潔にする

→ コードの変更時の影響範囲を小さくする



問題例3(1/2)

◆ 階層を跨ぐ依存関係がある



下位層から上位層への依存ではないが、設計時には階層を跨ぐ依存は意図していない

→ アーキテクチャの崩れ

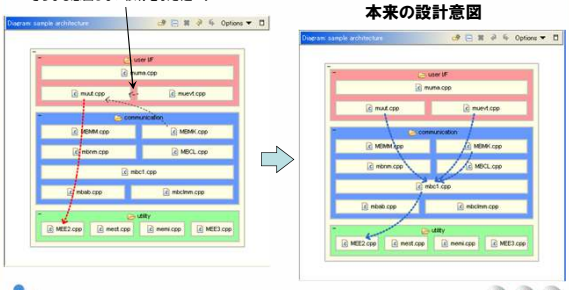
本来の設計意図

問題例3(2/2)

放置すると・・・

さらなる意図しない依存を引き起こす

本来の設計意図

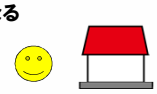


まとめ

アーキテクチャが崩れると、テストが大変になる

↓

崩れに気づくには、外から見ることが重要



↓

「グループ化」と「グループの上下関係」で設計図に近い形でコードの状態を可視化

↓

崩れたアーキテクチャを早い段階で修正

↓

アーキテクチャの維持

ご清聴ありがとうございました。

株式会社東陽テクニカ
ソフトウェア・ソリューション