既存コードの
グレイボックステスト
～潜在バグをあぶりだす～

発覚バグ

潜在バグ

ビースラッシュ株式会社

Copyright BACKSLASH DESIGN Co., Ltd

目次

1．グレイボックステストとは

- テストの原則
- 構造化の効用
- グレイボックスボックステストの有効性と要点
- 悪循環を断つためのグレイボックステスト

2．RiTMUS法とは

- コードの良し悪しを定量化し、欠陥リスクの高い個所を特定する
- RiTMUS法の適用に先立ち、AtScopeで図面化を行う

3．RiTMUS法の適用手順

- スマートガレージコードでの適用

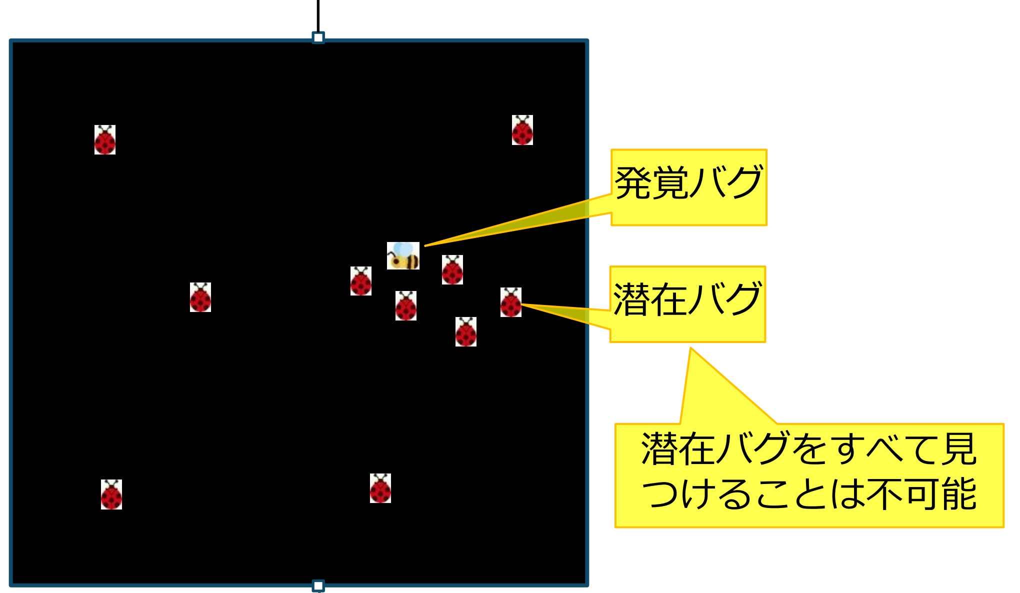
Copyright BACKSLASH DESIGN Co., Ltd

１．グレイボックステストとは

Copyright BACKSLASH DESIGN Co., Ltd

テストの原則

テストの限界	解説
バグが「ない」ことは示せない	テストでバグは検出できるが、バグが無くなったことは示せない。
全数テストは不可能	プログラムの分岐の数だけ、指数関数的にテストケースは増える。
バグゼロはたどり着けないゴール	千行を超えるコードは、完璧に作ることはできない。（形式手法で数百行であれば可能かもしれない）

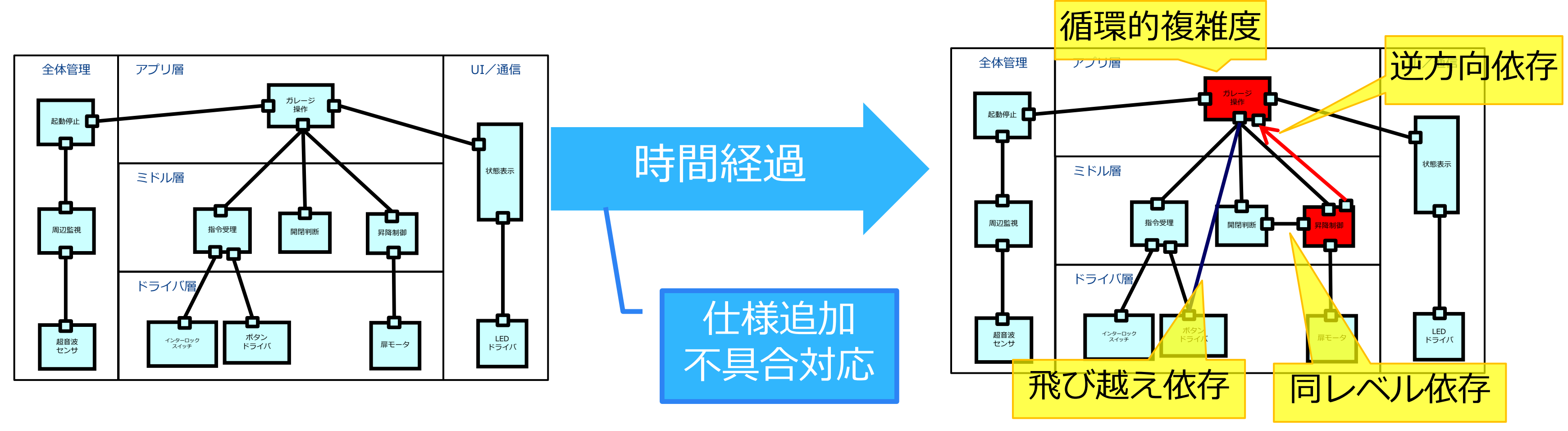


本資料でのアイコン

	発覚バグ	テストで見つけた欠陥
	潜在バグ	まだ見つけられていない欠陥

Copyright BACKSLASH DESIGN Co., Ltd

構造化の効用



構造化の効用	内容	ソフトウェア開発では
全体の俯瞰	構成要素を網羅して、全体像と各要素の位置づけを把握することができる。	開発分担や結合テストがしやすくなる。
関係性の把握	要素間の関係性を図面化し、構造的複雑度を把握できる。	変更による影響範囲が推測できる。
相対的な重要度	重要な要素を特定できて、それがなぜ重要なのかを把握できる。	効果的かつ効率的なテスト計画ができる。
時間的な因果関係	原因と結果の因果律が分かる。時間経過による変化とその要因を紐づける。	改善や再発防止ができる。

Copyright BACKSLASH DESIGN Co., Ltd

グレイボックステストの有効性

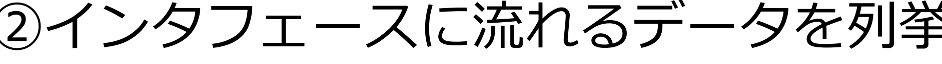
- 内部を構造化して、要素間のインタフェースをテストする
- 単純にテストケースを増やすのではなく、戦略的にテストケースの設計ができる

ブラックボックステスト

結合テスト
グレイボックステスト

Copyright BACKSLASH DESIGN Co., Ltd

グレイボックステストの要点



③テストケースを取捨・拡充

テストケース
テストケース

③テストケース
を取捨・拡充

欠陥リスクの高い要素
を集中的にテスト

②インタフェースに
流れるデータを列挙

①バグの多そうな
要素を特定

欠陥リスクの低い要素
のバグは検出できない

すべては検出できない

その他の個所は
検出できない

Copyright BACKSLASH DESIGN Co., Ltd

悪循環を断つためのグレイボックステスト

- ソフトウェア資産化と効果的効率的テストの両立

テスト工数とコードの複雑さが増大する → テスト工数が管理できてコードはシンプルになる

テストケースを追加 (大量)
↓
ブラックボックステストで不具合が発生 (バグ)
↓
局所的な修正でコードが複雑化 (バグの連鎖)

設計構造を理解 (資産)
↓
グレイボックステスト (高速)
↓
大局を見て修正 (高品質)

Copyright BACKSLASH DESIGN Co., Ltd

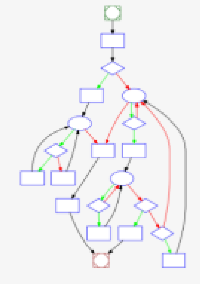
2．RiTMUS法とは

Copyright BACKSLASH DESIGN Co., Ltd

RiTMUS(リトマス)法とは

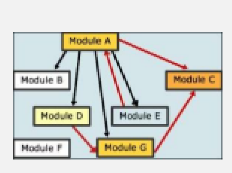
- Rits Test Metrics Utilize Style/Suiteの略
  - リコーITソリューションズが開発したリスクに基づくテスト設計技法
  - ソフトウェアの構造に基づきテストアプローチを導出する
  - ソースコードの複雑さを基にバグが発生するリスクを予測し、そのリスクを構造図にマッピングし、テスト範囲を決定する手法

循環的複雑度



循環的複雑度	区分
51～	高
21～50	中
11～20	低
1～10	無

構造的複雑度



構造的複雑度		区分
FanIn	FanOut	
High	High	高
Low	High	中
High	Low	低
Low	Low	無

欠陥リスク

循環複雑度					
	高	7	10	13	16
	中	5	9	12	15
	低	3	6	11	14
	無	1	2	4	8
		無	低	中	高
		構造的複雑度			

安全!

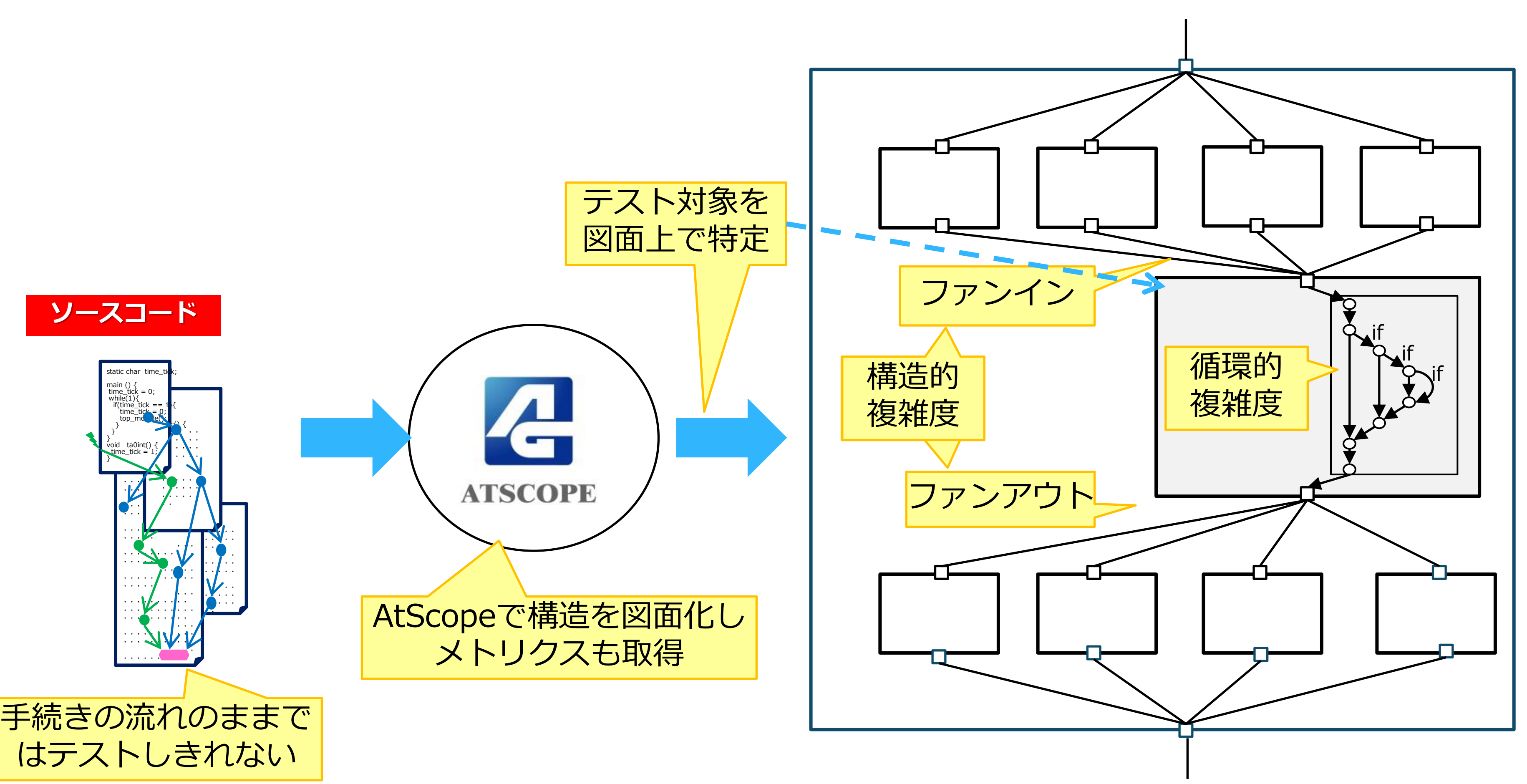
【欠陥リスク】
循環的複雑度と構造的複雑度から算出したバグが発生する予測値を、危険度として16段階の数値で表現します

循環複雑度、構造複雑度の図の出典元：
http://understand-jp.blogspot.com/2012/06/blog-post.html
https://www.techmatrix.co.jp/product/lattix/function/impactanalysis.html

Copyright BACKSLASH DESIGN Co., Ltd

構造図とメトリクス（構造的複雑度と循環的複雑度）

- ソースコードからAtScopedeで図面化し、メトリクスも計測



Copyright BACKSLASH DESIGN Co., Ltd

循環的複雑度の区分：サイクロマチック複雑度

- 循環的複雑さのメトリクスとして「サイクロマチック複雑度」がある
- サイクロマチック複雑度の算定式

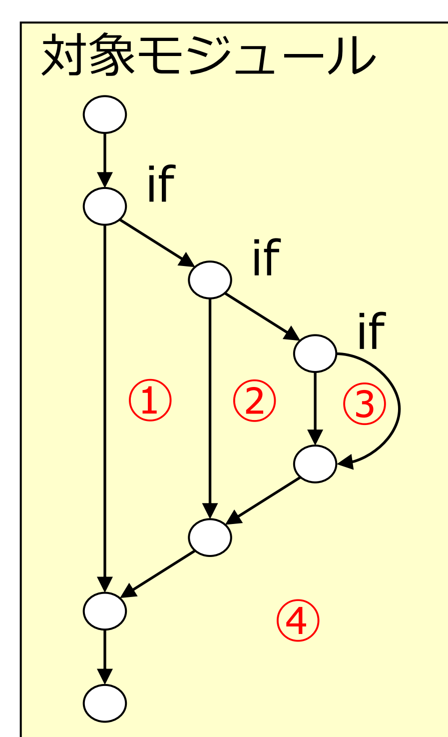
  複雑度C=e-n+2

  n=プログラムに含まれる分岐点の数（矢印の数）

  e=プログラムに含まれるノードの数（丸の数）

  - 右図の場合は、C=10-8+2=4

- 図解して閉空間の数を数えることもできる
  - 右図の場合は①～④で「４」　※外側も数える



- 複雑度の品質判定

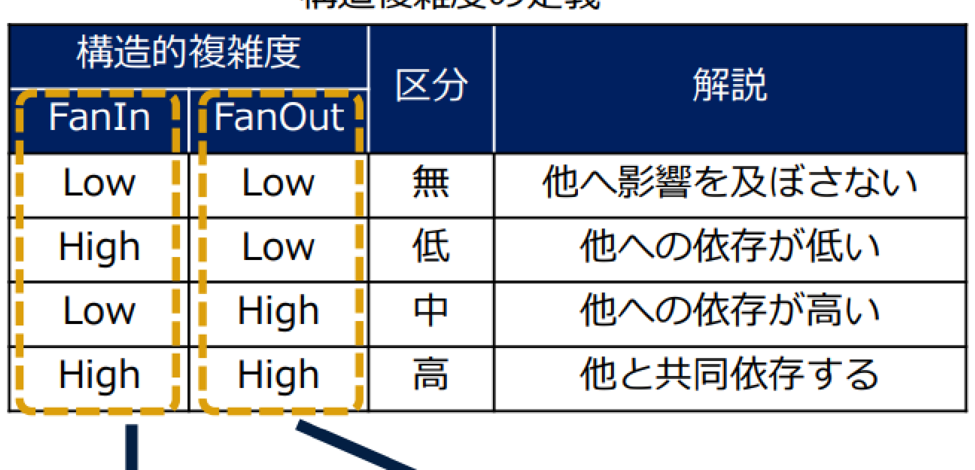
複雑度	区分	リスク評価
1～10	無	テストしやすい
11～20	低	少し複雑であるがテストはできる
21～50	中	複雑でテストが難しい
50以上	高	テスト不可能なプログラム

Copyright BACKSLASH DESIGN Co., Ltd

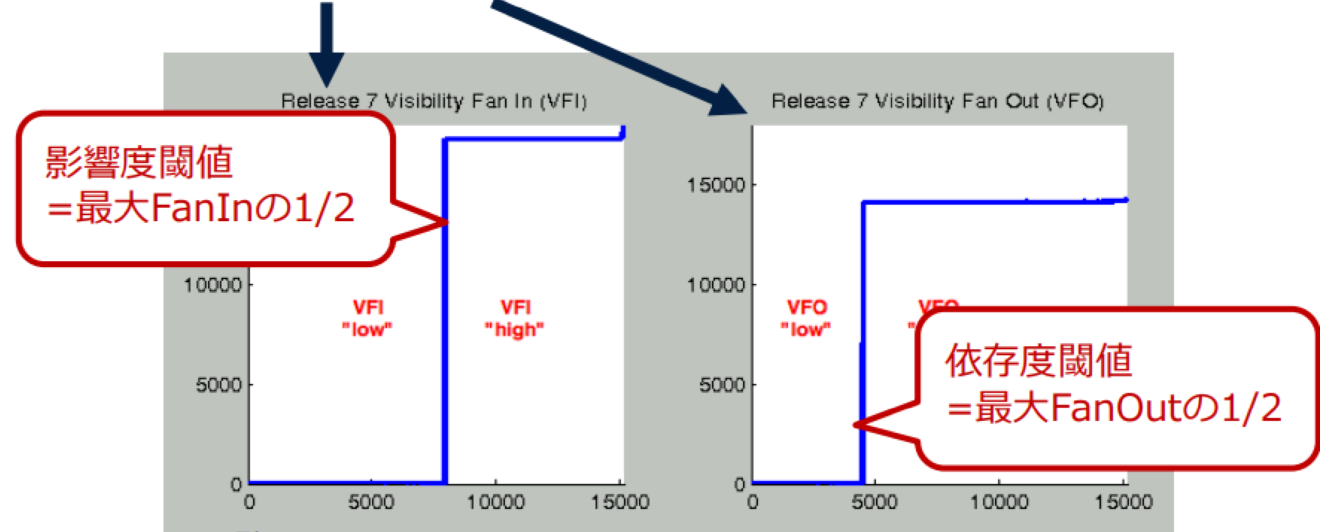
構造の複雑さを示す尺度

- 構造的複雑度の評価方法

構造複雑度の定義



構造的複雑度		区分	解説
FanIn	FanOut		
Low	Low	無	他へ影響を及ぼさない
High	Low	低	他への依存が低い
Low	High	中	他への依存が高い
High	High	高	他と共同依存する



引用：Dan Sturtevant, INCOSE Webinar June 12thm 2013, “Slide 22”

コンポーネントごとのFanIn・FanOutが、「閾値」と比べ高いか低いかにより判定する

Copyright BACKSLASH DESIGN Co., Ltd

３．RiTMUS法の適用手順

Copyright BACKSLASH DESIGN Co., Ltd

RiTMUS法の適用手順

ソース
ファイル
(.c)

構造図を
作成する
AtScope

欠陥リスク値
を算出する
RiTMUS

テスト対象
を特定する
AtScope
RiTMUS

インタフェース
を列挙する
AtScope

テストケース
を補強する

テスト
ケース

Copyright BACKSLASH DESIGN Co., Ltd

コードから構造図を作成する

- AtScopeで図面化する

フォルダを指定して「実行」

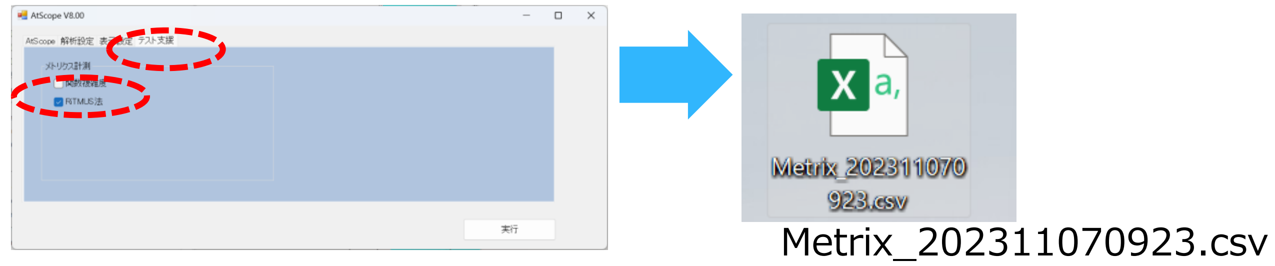
アーキテクチャ形状に配置（A2G配置）

配置

Copyright BACKSLASH DESIGN Co., Ltd

欠陥リスク値を算出する

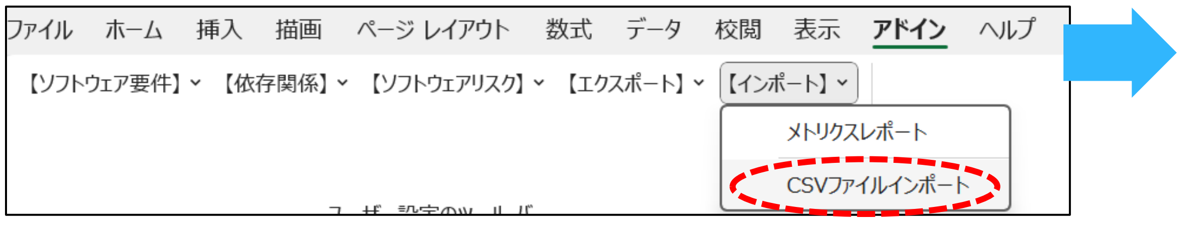
- AtScopeでcsvメトリクスファイルを生成
  - 「テスト支援」タブ→「RiTMUS法」をチェック



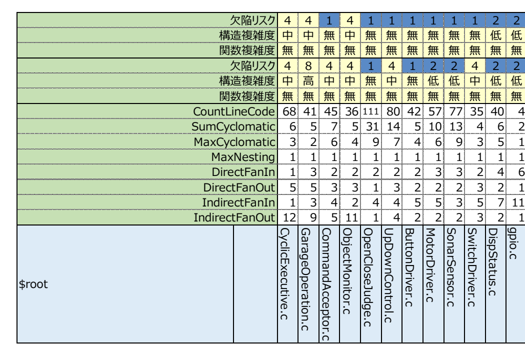
Metrix_202311070923.csv

- RiTMatrix_V100.xlsmでcsvメトリクスファイルをインポート
  - アドイン→[インポート]→csvファイルインポート

RiTMatrix_V100.xlsm



欠陥リスク（１～16段階）

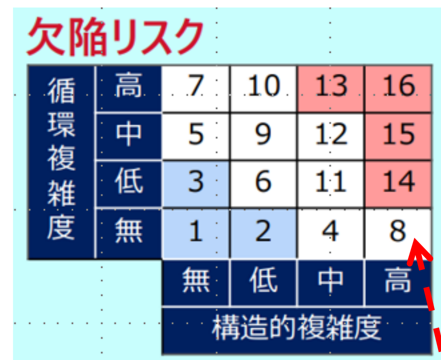


$root	CyclicExecutive.c	GarageOperation.c	CommandAcceptor.c	ObjectMonitor.c	OpenCloseJudge.c	UpDownControl.c	ButtonDriver.c	MotorDriver.c	SonarSensor.c	SwitchDriver.c	DispStatus.c	gpio.c
欠陥リスク	4	4	1	4	1	1	1	1	1	1	2	2
構造複雑度	中	中	無	中	無	無	無	無	無	無	低	低
関数複雑度	無	無	無	無	無	無	無	無	無	無	無	無
欠陥リスク	4	8	4	4	1	4	1	2	2	4	2	2
構造複雑度	中	高	中	中	無	中	無	低	低	中	低	低
関数複雑度	無	無	無	無	無	無	無	無	無	無	無	無
CountLineCode	68	41	45	36	111	80	42	57	77	35	40	4
SumCyclomatic	6	5	7	5	31	14	5	10	13	4	6	2
MaxCyclomatic	3	2	6	4	9	7	4	6	9	3	5	1
MaxNesting	1	1	1	1	1	1	1	1	1	1	1	1
DirectFanIn	1	3	2	2	2	2	2	3	3	2	4	6
DirectFanOut	5	5	3	3	1	3	2	2	2	3	2	1
IndirectFanIn	1	3	4	2	4	4	5	5	3	5	7	11
IndirectFanOut	12	9	5	11	1	4	2	2	2	3	2	1

Copyright BACKSLASH DESIGN Co., Ltd

欠陥リスク値からテストすべき要素を特定

- 欠陥リスク値が高い要素を特定
  - 特に13～16は危険！



循環複雑度					
	高	7	10	13	16
	中	5	9	12	15
	低	3	6	11	14
	無	1	2	4	8
		無	低	中	高
		構造的複雑度			

欠陥リスクの
最大値は「８」

その要素はGarageOperation

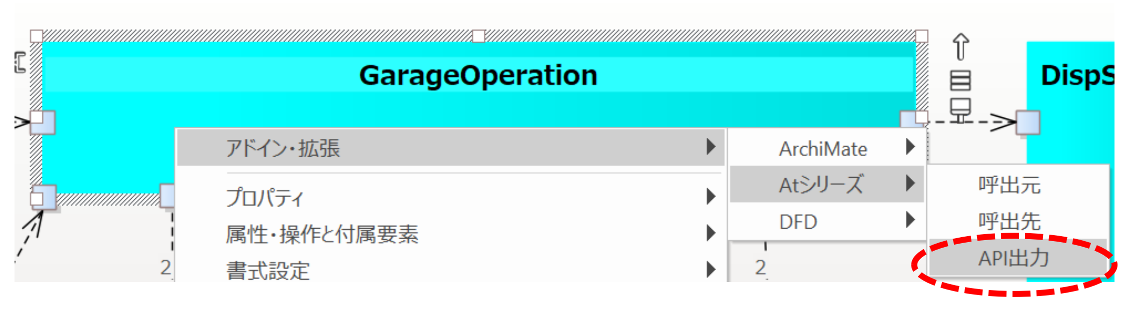
$root	CyclicExecutive.c	GarageOperation.c	CommandAcceptor.c	ObjectMonitor.c	OpenCloseJudge.c	UpDownControl.c	ButtonDriver.c	MotorDriver.c	SonarSensor.c	SwitchDriver.c	DispStatus.c	gpio.c
欠陥リスク	4	4	1	4	1	1	1	1	1	1	2	2
構造複雑度	中	中	無	中	無	無	無	無	無	無	低	低
関数複雑度	無	無	無	無	無	無	無	無	無	無	無	無
欠陥リスク	4	8	4	4	1	4	1	2	2	4	2	2
構造複雑度	中	高	中	中	無	中	無	低	低	中	低	低
関数複雑度	無	無	無	無	無	無	無	無	無	無	無	無
CountLineCode	68	41	45	36	111	80	42	57	77	35	40	4
SumCyclomatic	6	5	7	5	31	14	5	10	13	4	6	2
MaxCyclomatic	3	2	6	4	9	7	4	6	9	3	5	1
MaxNesting	1	1	1	1	1	1	1	1	1	1	1	1
DirectFanIn	1	3	2	2	2	2	2	3	3	2	4	6
DirectFanOut	5	5	3	3	1	3	2	2	2	3	2	1
IndirectFanIn	1	3	4	2	4	4	5	5	3	5	7	11
IndirectFanOut	12	9	5	11	1	4	2	2	2	3	2	1

class 解析結果
CyclicExecutive
GarageOperation
DispStatus
CommandAcceptor
OpenCloseJudge
UpDownControl
ObjectMonitor
SwitchDriver
ButtonDriver
MotorDriver
SonarSensor
gpio

Copyright BACKSLASH DESIGN Co., Ltd

インタフェースを列挙してテストケースを補強する

- 特定された要素のインタフェースを管理する
  - EAで要素を右クリック→Atシリーズ→API出力でcsvファイル出力



インタフェースの引数を
データ辞書で定義して、
テストケースを作成する

GarageOperation_ifDef_20231107.csv

No	check	AtScope/ファイル構造図/GarageOperation	引数	戻り値	副作用	呼出元ファイル	呼出元関数	呼出先ファイル	呼出先関数	呼出先変数
1	Not	CyclicExecutive.app_main->GarageOperation.go_init				CyclicExecutive	app_main	GarageOperation	go_init	
2	Not	CyclicExecutive.app_main->GarageOperation.go_run				CyclicExecutive	app_main	GarageOperation	go_run	
3	Not	GarageOperation.go_run->CommandAcceptor.ca_notifyCommand				GarageOperation	go_run	CommandAcceptor	ca_notifyCommand	
4	Not	GarageOperation.go_init->CommandAcceptor.ca_init				GarageOperation	go_init	CommandAcceptor	ca_init	
5	Not	GarageOperation.go_run->OpenCloseJudge.oj_judgeAction				GarageOperation	go_run	OpenCloseJudge	oj_judgeAction	
6	Not	GarageOperation.go_init->OpenCloseJudge.oj_init				GarageOperation	go_init	OpenCloseJudge	oj_init	
7	Not	GarageOperation.go_run->UpDownControl.uc_operateDoor				GarageOperation	go_run	UpDownControl	uc_operateDoor	
8	Not	GarageOperation.go_init->UpDownControl.uc_init				GarageOperation	go_init	UpDownControl	uc_init	
9	Not	GarageOperation.go_init->DispStatus.ds_init				GarageOperation	go_init	DispStatus	ds_init	
10	Not	ObjectMonitor.om_monObject->GarageOperation.go_notifyDetection				ObjectMonitor	om_monObject	GarageOperation	go_notifyDetection	
11	Not	ObjectMonitor.om_monObject->GarageOperation.go_notifyRemoved				ObjectMonitor	om_monObject	GarageOperation	go_notifyRemoved	

Copyright BACKSLASH DESIGN Co., Ltd

参考

Copyright BACKSLASH DESIGN Co., Ltd

アーキテクチャテンプレートA2G配置

- 水平垂直分割して全体を俯瞰
- 全体を俯瞰し戦略的な開発へ

機能実現部
3層に分ける

全体管理
(**a**dmin)

アプリ層(**b**oss)

ミドル層(**c**trl)

ドライバ層(**d**rv)
エンティティ層(**e**tty)

ユーザ
インタフェース
や
通信
(**f**acade)

実行基盤層(**g**nd)

全体に影響を及ぼす横断的関心
・初期化
・エラー
・機内監視
など

異なる問題ドメイン
・UI
・通信
など

実行環境の依存部
OSライブラリなど

Copyright BACKSLASH DESIGN Co., Ltd

データ辞書の表記法

表記例	説明	名称
＝	＝の左辺はデータ、右辺はデータの意味	等号
X＝型，値域，単位	Xは、指定の値域で任意の値を取る	変数
X＝a ＋ b	Xは、aとbを常に要素として持っている	連結
X＝[a | b]	Xは、aまたはbのいずれかである	選択
X＝n{ a }m	Xは、aのn回からm回の繰り返しである	繰返し
X＝“ABC“	Xは、文字列 “ABC”である	リテラル
X＝*説明文*	Xは、*説明文*という意味である	コメント

Copyright BACKSLASH DESIGN Co., Ltd