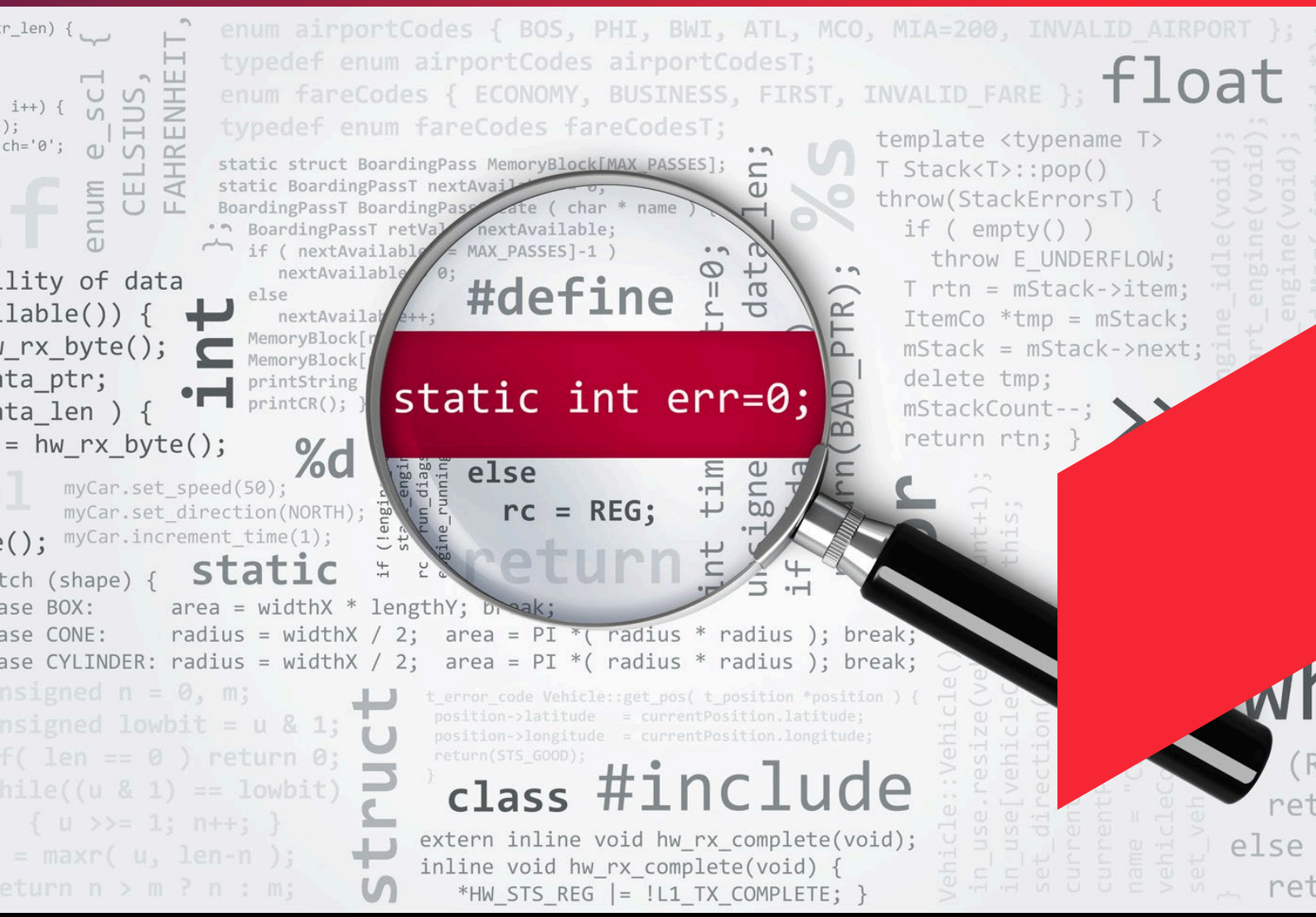
VECTOR >

はじめての単体テスト

ビギナー向け資料「はじめてシリーズ」



目次

発行元：ベクター・ジャパン株式会社

はじめに

本資料では、「単体テストとは何か？」、「なぜ単体テストが必要なのか？」、「どのようにすれば効率的に単体テストを行うことができるのか？」といった観点から、近年の組み込み業界の現状や単体テストについての基本的な知識をわかりやすく説明していきます。

1. 単体テストと機能安全

1.1 組み込み業界の現状

過去数十年にわたって、ソフトウェアは自動車システムの設計において重要な役割を果たしてきました。自動車は、ハードウェアベースからソフトウェアベースへと変化し続けています。近年のマッキンゼーの調査 [1]によれば、2010 年には 1 車両に対して約 1000 万行のコード数であったのが、2016 年にはその数が 15 倍に増加して 1 億 5000 万行ものコード数に達しているとのことです。

こうした背景のなか、各自動車メーカーはソフトウェアの機能で市場での差別化を図っています。マッキンゼーは、ソフトウェアがどのように自動車の進歩を実現しているかの例として、コネクティビティ、自動運転、電動化、多様なモビリティ、の 4 つの言葉を挙げています。これらすべての機能において、ソフトウェアの安全性と品質が大きな役割を果たすことは言うまでもありません。

要件が厳しく、複雑なシステムになっていくなかで、開発コストをどう低減していくかは自動車ソフトウェア開発における重要な課題となっています。また、実装工数が増えることでテスト工数も増えることとなり、より効率的なテスト環境を構築することが求められています。

また、ADAS（先進運転支援システム）や自動運転などの安全に関わる分野では、製造者の責任が問われる可能性があります。テストを行うだけでなく、テストを行ったことをドキュメント化していくことも求められており、信頼性のあるツールを使用することが重要になってきています。

1.2 単体テストとは何か？

ひと口に「単体テスト」といっても、会社によって意味するところは少しずつ違います。本資料では、関数単位で行うテストを「単体テスト」と定義します。

単体テストでは、ホワイトボックステストやブラックボックステストといったテスト手法を使って、ソフトウェアが要求どおりに動作するか、不具合を発生させても異常動作とならないか、コードカバレッジが 100%となるか、などをテストします。

単体テストは実機上で行うこともできますが、実機の代わりにテストツールを使用することで、実機に近いシミュレーション環境でテストを行うことができます。この手法については、自動車分野の機能安全規格 ISO 26262 にも定義されています。

---

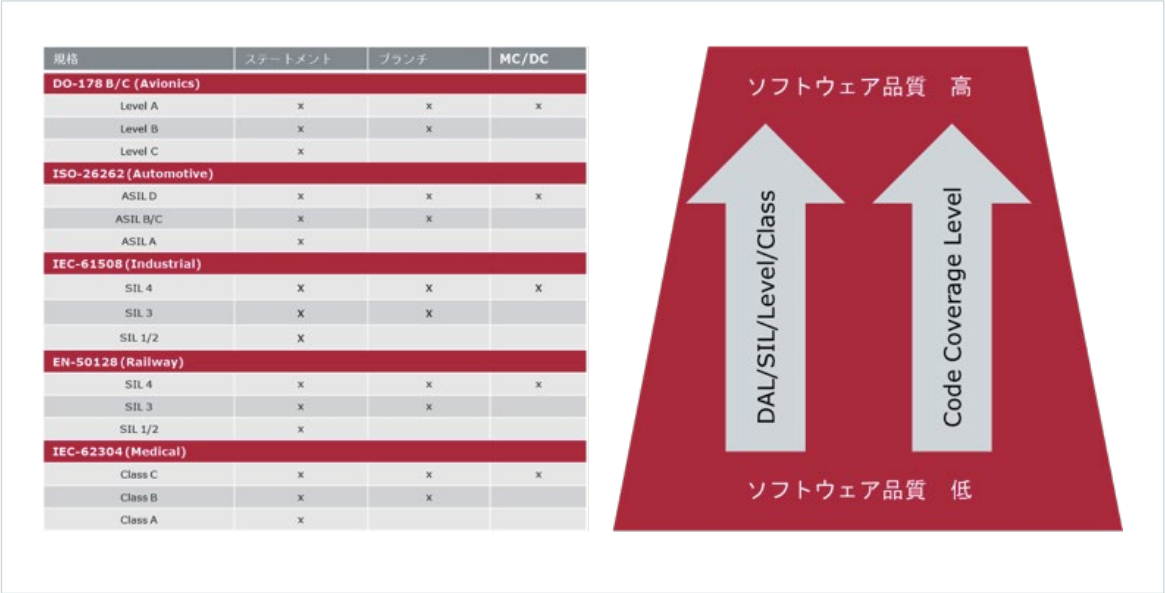
[1] McKinsey & Company；「Rethinking car software and electronics architecture 」；Burkack y、Deichmann、Doll、Knochenhauer；2018 年 2 月（参照：弊社資料「Following the ISO 26262-6:2018 Guidelines with VectorCAST」；Whitepaper | V1.1 2019-10）

1.3 単体テストと機能安全認証との関係性

『自動車 – 機能安全』というタイトルが付いた ISO 26262 は、自動車に搭載される電気／電子システムの機能安全に関する国際規格として、国際標準化機構（ISO）によって 2011 年に定められました。ISO 26262 規格の第 6 部には、ソフトウェア開発規格の一部として、動的なソフトウェアテスト・検証に関する推奨事項が記載されています。

推奨される活動には、関数テストなどのユニットレベル・統合レベル・システムレベルのテスト（要件ベースのテストとパーティションテスト）と、構造的カバレッジテストの両方が含まれます。各種認証と、そこに求められるカバレッジレベルの関係は図 1 のような構成になっています。ここで、図1で使用している DAL、SIL、ASIL とは、以下の指標を表します。

- DAL = Design Assurance Level。DO-178 B/C で用いられる
- SIL = Safety Integrity Level。IEC-62508 や IEC-61508 で用いられる
- ASIL = Automotive Safety Integrity Level。ISO-26262 で用いられる



規格	ステートメント	ブランチ	MC/DC
DO-178 B/C (Avionics)			
Level A	x	x	x
Level B	x	x	
Level C	x		
ISO-26262 (Automotive)			
ASIL D	x	x	x
ASIL B/C	x	x	
ASIL A	x		
IEC-61508 (Industrial)			
SIL 4	x	x	x
SIL 3	x	x	
SIL 1/2	x		
EN-50128 (Railway)			
SIL 4	x	x	x
SIL 3	x	x	
SIL 1/2	x		
IEC-62304 (Medical)			
Class C	x	x	x
Class B	x	x	
Class A	x		

図 1: 各種認証と求められるカバレッジレベルの関係

ソフトウェアエラーの深刻度が高いほど、ソフトウェア品質に対する要求が高くなり、必要なカバレッジレベルが上がります。テストケースによって達成されるコードカバレッジ率が高ければ高いほど、期待されるソフトウェアの品質は向上すると言われていますので、ソフトウェアの品質を向上させるためにはコードカバレッジ測定が非常に有効です。

また、ISO 26262 では、プログラムが稼動する実環境にできるだけ近い環境で統合テストケースを実行することを強く推奨しています。

ISO 26262 には、自動車の安全関連プロジェクトに使われるソフトウェアツールの認定に関する要求が含まれています。ツールの適正な扱いを定めるために、以下の 2 ステップのアプローチが使われます。[2]

ステップ 1: 分類

発生する可能性のある安全目標侵害、および発生する可能性のあるフォールトを検出できる確率に基づき、信頼水準を決定する

これが安全関連プロジェクトでツールを使用する場合の前提条件となる

ステップ 2: 認定

要求される信頼水準に基づき、ツールを認定するための方策を定義する

各ソフトウェアツールでは、要求される信頼水準（Tool Confidence Level; TCL）を分析することが求められています。もし、単体テストツールに対してツール認証が必要と判断された場合でも、機能安全認証を取得している単体テストツールを使用すれば、あらためてツール認証を取得する必要はありません 。

あくまで、機能安全はベストプラクティスであり、ツール認証を取得していないツールで単体テストを行っていたり、クロスコンパイルしていない環境でテストを行っていたりする場合もあります。しかし、ベストプラクティスとしては認証を取得している単体テストツールの使用が推奨されます。

[2] 弊社資料「ベクターのツールと機能安全 ISO 26262；2019 年 8 月 5 日」より抜粋

2. なぜ単体テストが必要なのか？

2.1 なぜ単体テスト工程は飛ばされてしまうのか

自動車のソフトウェア開発に携わっていて、ISO 26262 に準拠した開発を行っている方々は単体テストを実施していると思いますが、ほかの組み込み業界では単体テストを行っていない会社が多いのが現状です。なぜ、単体テスト工程は飛ばされてしまうのでしょうか。

その要因として考えられるのは、単体テストには非常に時間が掛かるという点です。また、単体テストが実施されていない膨大なレガシーコードに対してテストをするためのコストを試算すると、余りにもコストが高く、結局は単体テストを行わないという判断になってしまうことも考えられます。

単体テストに時間が掛かっている場合は、古典的な単体テストを行っている可能性があります。たとえば、テストドライバーやスタブのコードを書いていたり、無償のテストツールを使用していたりすることにより自動化が進んでいないことが背景にあるのかもしれません。

こうした、工数が掛かるという問題に対しては、自動化をサポートする有償の単体テストツールの使用が有効な解決法だといえます。ツールを導入したうえで、まずは新しく追加する機能に対して単体テストを行っていくことが重要です。さらには、システムの中で複雑だと思われる部分に対して単体テストを行うことも効果的です。いずれにしても、まずは小さい範囲から単体テストツール導入の効果を測定していくことをお勧めします。

多くの開発者が「単体テストを行いたい」という気持ちを持ってはいるものの、単体テストを行うためには開発のワークフロー自体を変えていく必要があり、そのことが単体テストツール導入の妨げになっているようです。また、開発しなければならないソフトウェアが増えていく一方で、社内のソフトウェア開発者数は増えないといった問題もあるでしょう。社内のリソース不足の問題が、新たなツール導入の妨げとなっているとも考えられます。

開発のワークフローという点で見ると、日本では開発部門と評価部門が分かれている企業が多く、実装をすべて終えた後に単体テストフェーズへ移行するという会社が多いのですが、効率的なテストを行うためには、実装してすぐにテストを行うことが理想的です。海外の自動車メーカーでは、実装後に細かい単位で単体テストを行うことが求められるケースも出てきています。

自動車の開発現場では、ソフトウェアの評価を専門に行う部署を立ち上げている先進的な企業もあります。このようなソフトウェアのテストを行う仕組みを会社として作っていくことが必要になってきていることがうかがえます。

2.2 なぜ単体テストを行うべきなのか

単体テストは、一般に、全テスト工程の中でテスト対象とするソースコード量が最も少ないテストとなります。そのため、不具合を発見した場合にソースコードの不具合箇所を特定しやすく、修正に掛かる時間が短くて済む点が利点として挙げられます。

当然のことながら、単体テストで不具合を検出することができれば、後工程のテストフェーズでの不具合発生を防ぐことができます。不具合発見が後の工程になればなるほど、不具合が開発中のソフトウェアに与える影響範囲が大きくなり、手戻りが増えることになります。

単体テストで不具合を発見した場合と、結合テストやシステムテストの段階で不具合を発見した場合では、不具合の修正に掛かるコストは大きく違ってきます。図2は IBM による調査、図3は Google による調査、図 4 は複数プロジェクトの分析の要約（約 250 プロジェクト）です。各テストフェーズで、大きくコストが違うことがわかります。

ソフトウェア開発ステージ	バグ修正にかかるコスト*
設計	1x
実装	7x
テスト	15x
保守	100x

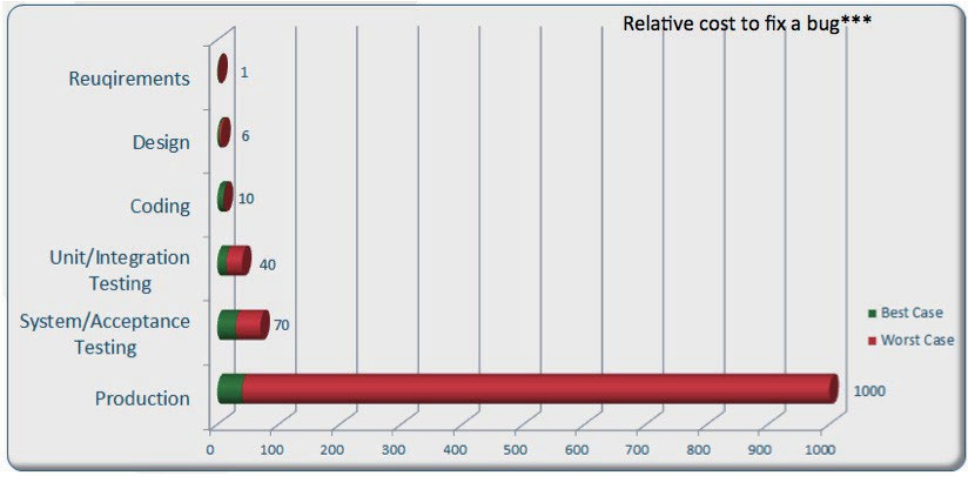
* DevOps: Shift left with continuous testing by using automation and virtualization (IBM; Dibbe Edwards)

図 2: 各ソフトウェア開発ステージにおける、バグ修正にかかるコストの対比

テストフェーズ	~コスト/バグ**
単体テスト	$5
フルビルド	$50
結合テスト	$500
システムテスト	$5000

** How Google Tests SW (James A. Whittaker, Jason Arbon, and Jeff Carollo)

図 3: 各テストフェーズにおける、バグ修正にかかるコストの対比



*** James Martin, An Information Systems Manifesto, Prentice-Hall, Inc., Englewood Cliffs, New Jersey

図 4: 要求仕様作成から生産までの各ステージにおける、バグ修正にかかるコストの対比

自動車業界では、不具合の検出コストよりも、リコールに伴う直接コストや製品ブランドへのダメージの方が影響として大きいため、徹底的なテストを行うことが必要不可欠となっています。そのことからも、単体テストで不具合を検出することが重要です。

2.3 レガシーコードへの対応

単体テストを導入する際は、テストインフラが不十分であった時代に作られたレガシーコードに対し、どうテストを行っていくかを考える必要があります。社内のコード規約などが整備されていない時代に作られたレガシーコードを、リファクタリングせずに使い続けているケースも見受けられます。「うちのレガシーコードは改善しようがない」という声をお聞きすることがありますが、こうした姿勢では品質向上を叶えることは難しくなります。品質志向に移行するためには、テストは時間と手間がかかるものと認識し、取り組んでいかなければならないのですが、開発者のリソースは限られており、開発と並行してレガシーコードへのテスト環境構築を行うことは現実的ではありません。テストインフラを整えるためには、会社の方針としてリソースを割り当てる必要があります。たとえば、ベテランのチームメンバーでグループを組み、単体テストの導入を先導することや、受託会社などを利用して単体テストを依頼するなど、テストインフラを整えるための費用を捻出する必要があります。

レガシーコードは多くの場合、技術的負債を抱えていて、解決するためにはリファクタリングを行うか、レガシーコードのアプリケーションの正確性を確認するための回帰テスト環境を構築するしかありません。もちろん、どちらも非常に工数と費用が掛かる作業です。リファクタリングを行うことは影響範囲の確認を随時行う必要があるということなので、リファクタリングを行う際も回帰テストの環境を構築することが必要となります。ガートナーの調査 [3]には、「反復可能なテストケースがないと、客観的かつ測定可能な方法で機能等価性を実証しにくい」と記されています。

重要なことは、開発・テストのワークフローとメトリクスを設定し、ソフトウェア品質を高めるベストプラクティスを考えていくことです。その一つとして、回帰テスト環境の構築があります。日本の自動車業界のほとんどのお客様がカバレッジを取得するテストケースを作成しており、カバレッジ取得 100%を達成していますが、回帰テスト環境の構築を行っている会社は非常に少ないのが現状です。一方、海外の自動車開発業界では、回帰テスト環境を実現している会社が多くあります。この流れは、アメリカのソフトウェア企業が以前からテスト自動化環境を構築していたことも要因として考えられます。

[3] Gartner Group;「Monitor Key Milestones When Migrating Legacy Applications」;2015 年 5 月 18 日(参照:弊社資料「ベースラインテスト:レガシーコードベースの技術的負債を減らす秘訣」;ホワイトペーパー | V2.0 2018-01)

3. 単体テストの自動化

3.1 Google のソフトウェア開発での例

先進的なソフトウェア企業では、どのようなテスト自動化環境を構築しているのでしょうか。 Google は、ソフトウェア開発でどのようなところにコストが掛かっているかを試算しています。そのデータの一部から計算方法を利用して、中規模の自動車開発プロジェクトに年単位で当てはめた場合の例をご紹介します。図 5[4] を見ると、チームが不具合の修正に費やしているスタッフの割合は 40%に達しています。Google の試算結果では、開発での 30%から 50%の時間が不具合修正に費やされていることになります。

1年間に生成されるソースコードの行数(KSLOC)		200
1000 SLOCあたりの平均バグ数[2]	x	8
コード内のバグ数	=	1,600
1つのバグの修正に要する平均コスト[3]	x	$1,500
バグ修正に要する年間合計コスト	=	$2,400,000
エンジニアの年間コスト[4]	/	$150,000
バグ修正に要するエンジニアの人数	=	16
エンジニアのチームのサイズ	/	40
バグ修正に投入されるスタッフの割合		40%

図 5: Google のメトリック

Google は、「コード内に不具合が出るのは当たり前」という考え方に基づき、テストをできるだけ自動化して、不具合を開発プロセスのできるだけ早い段階で発見することを目標にしています。そのため、実装後はできるだけ迅速に自動テストが可能な単体テストケースの構築を行っています。これは、それぞれのテストフェーズで不具合を発見した場合のコストが違うことを理解しているからです。テストプロセスを改善し、自動化すれば、全体的な保守コストが削減できます。Google は、コード変更ごとにテストケースが実行されるようテストプロセスを構築しており、1日に 1.2 億のテストケースを自動実行しています。テストケースの作成を資産化し、効率よくテストを行うお手本のような企業だといえます。

Google だけでなく、Salesforce や Amazon などのアメリカのソフトウェア企業は、自社でテスト自動化を行う環境構築に大きな投資を行っています。自動車業界でも、今後さらにソフトウェアの規模が大きくなっていくなかで、先行してテストを自動化するための仕組みへの投資を行うことが求められます。

最初の一歩としては、静的解析ツールと単体テストツールの導入、そして、自動化を行うことができる CI ツール、たとえば Jenkins の導入などをお勧めします。また、テストを自動化するだけでなく、テストレポートのドキュメント化や、テストプロセスや結果の可視化ができるようになるとよいでしょう。

[4] 弊社資料「バグの修正および防止コストの定量化」; ホワイトペーパー | V2.0 2019-04 より抜粋
https://www.vector.com/jp/ja/know-how/vj-docs-dl/vj-whitepapers/quantifying-the-cost-of-fixing-vs-preventing-bugs/

3.2 回帰テスト自動化環境の導入メリット

新しい機能を追加することは、コードの増加を意味します。また、コードのメンテナンス費用も増加するでしょう。前述のとおり、新しい機能を追加するたびにコードをテストする仕組みが必要となってきます。

ここで、ソフトウェアテストにつきまとうコード変更や手戻りの問題について考えてみましょう。テストを進めていけば、不具合が見つかり、ソフトウェアを修正する必要が出てきます。また、仕様の変更が入ることにより、ソフトウェアが変更されることもあるでしょう。ソフトウェアに変更があるたびにテストをすべて初めからやり直すことは理想的ですが、実際には、ある程度の変更が行われた段階でまとめて再テストが実施されます。

この時に、どのコードの変更が不具合を起こしたのかが不明な場合は、修正前の状態に立ち返って修正を一つずつ検査する必要が生じてしまい、この手戻りが開発期間を長引かせる大きな要因となります。継続的インテグレーション（Continuous Integration; CI）手法では、変更を抱え込まず、変更のたびにテストすることにより、こうした手戻りを防止します。

また、修正からビルド、テストへの流れを自動化することで、品質の維持やコスト削減などの効果が出ます。

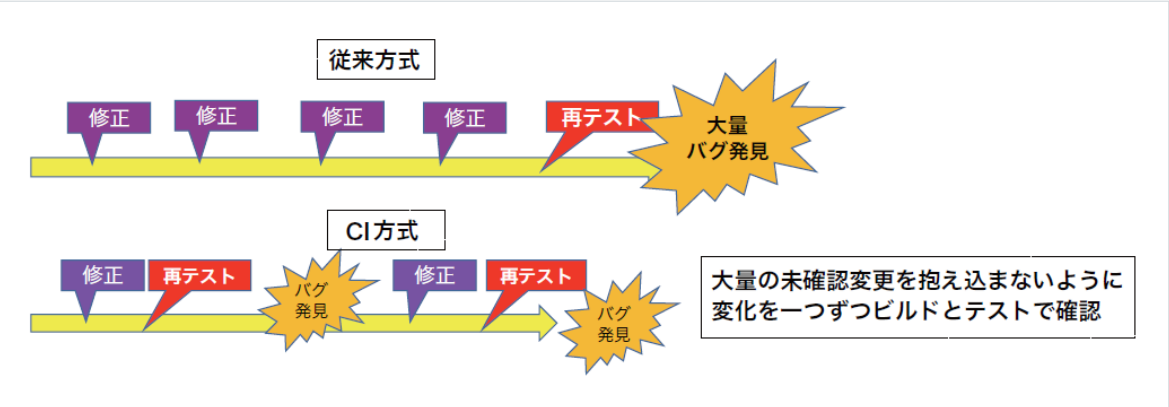


図 6: 未確認変更を抱え込まず、変更のたびにテストを行って手戻りを防止

組み込みシステムでは、ビルドの自動化システムの構築に手間が掛かったり、単体テストを自動化することが難しかったりといった問題がありました。CI 方式のテストを自動化するためには、以下の 3 点を自動化することが求められます。

1. コード変更で影響を受けるテストケースを判別
2. それらのテストケースをコード変更に対して実施
3. テスト結果のレポートを作成し、コードカバレッジを更新

この工程をツールで自動化することで、コーディング／テストサイクルを数日から数分に短縮することができます。

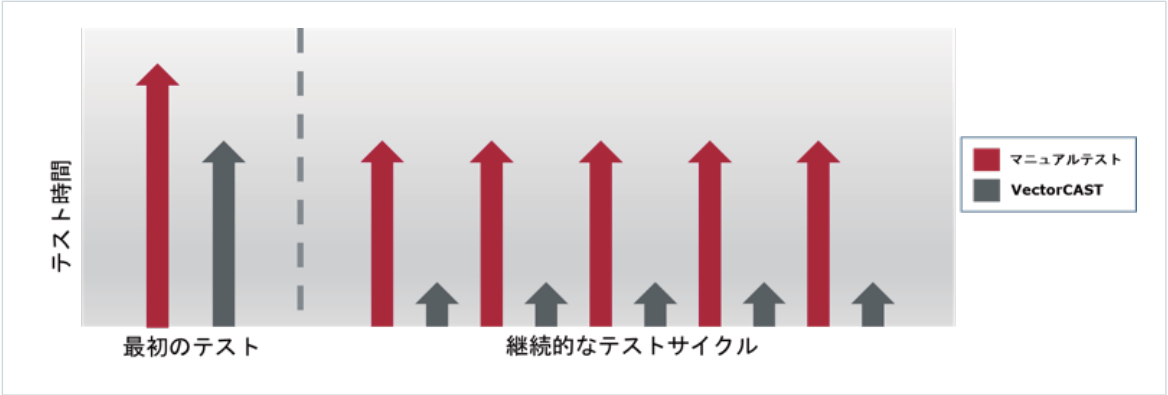


図 7: 最初のテストには時間が掛かるが、継続的にテストを行うことでテスト時間が大きく短縮可能

3.3 CI（継続的インテグレーション）を導入するためには

CI（継続的インテグレーション）を導入するためには、まず現状のテストプロセスがどのような状況かを把握することが重要です。単体テストといえば、カバレッジ 100%を達成すればよいと考える方もいらっしゃいますが、カバレッジは分岐条件を満たしているかを把握するだけで、単体機能の不具合がないことを意味するわけではありません。

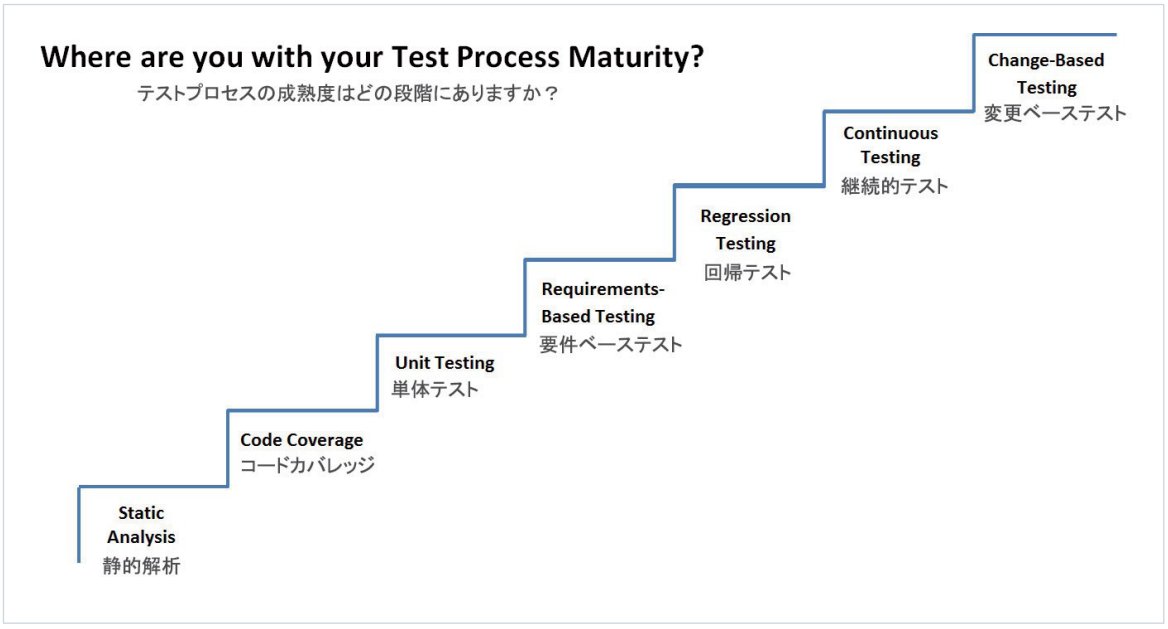


図 8: 現状のテストプロセスの把握

単体機能の不具合がないことを確認するために、入出力値のテストや要件ベースのテストを行います。

テストは一度行ったら終わりではありません。新しいコードを追加したら、すでにテストが終わっている箇所で不具合が発生していないかを、回帰テストにより確認します。また、新しいコードを追加したかどうかにかかわらず、継続的にテストを繰り返すことで、バグ発生の可能性をさらに減らすことができます。

回帰テスト以降のテスト段階では、過去に行ったテストも再度実行するため、回帰テストに対応したツールを使用してテストを自動化しておくと、効率的に開発を進めることができます。

4. ソフトウェアの品質管理

4.1 アジャイル型、DevOps の開発への転換

現代のソフトウェア開発はウォーターフォール型から、アジャイル型へと転換していますが、自動車業界ではまだ、古典的なウォーターフォール型の開発が一般的のようです。

アジャイル型の開発は、イテレーションとよばれる、細かい単位に区切った開発を繰り返していくことで進められます。1 イテレーションにつき 2～4 週間という短い期間の中で「計画→設計→実装→テスト」を行います。たとえば、Google では開発者が実装・テストを請け負っており、テストだけを担当するという人員は非常に少なくなっています。すべての開発者がテストツールを持ち、実装・テストの業務を、責任を持って請け負っています。自動車開発では Google の例と違い、無償のツールを使用しているわけではないため、開発者全員にテストツールを割り当てることはなかなか難しい現状があります。しかし、アメリカの自動車サプライヤーでは、ベクターの「VectorCAST」を開発者全員に割り当てるという取り組みが始まっています。

ソフトウェア規模の拡大は機能の増加を意味し、それはプロジェクトの途中での要求や仕様変更の増加をも意味します。現状のウォーターフォール型の開発では、プロジェクト途中での要件や仕様変更に弱いという問題点があります。特に、自動運転開発といった分野ではアジャイル型の開発を進めている企業がすでに見受けられ、他の分野の開発においても、アジャイル型の開発に移行していくことによってソフトウェアのリリース速度と品質を高めるという流れにつながっていくと考えられます。

次に、単体テスト工程における DevOps の考え方について考えてみましょう。

自動車開発では、先行開発部署と量産部署が分かれています。会社の規模や開発規模が大きくなればなるほど、この傾向は大きくなります。新しい機能を追加しても、実際の量産開発ではその機能をうまく活用できないという問題が起きます。こうした問題に対処するためには、DevOps の考え方が必要です。

DevOps とは、開発チーム（Development）と運用チーム（Operations）がお互いに協調し合うことで、開発・運用するソフトウェア／システムによってビジネスの価値をより高めるだけでなく、そのビジネスの価値をより確実かつ迅速にエンドユーザーに届け続けるという概念です。

自動車開発における問題点として、先行開発部署でしっかりとしたテストがなされていないということがあります。望ましいのは、新しい機能を追加すると共に、そのソフトウェアに対して自動化できるテスト環境を構築することです。テストを行ったドキュメントを残すことで、量産部署との対立を少なからず減らすこともでき、また、量産時のテスト効率を上げることもできます。新しい機能を開発するだけでなく、そのソフトウェアをテストする環境の構築が求められているのです。

5. 単体テストの手法

前に述べたように、単体テストは手作業で実施すると非常に工数が掛かるため、その工程ごと飛ばされてしまうことがあるほど、とかく敬遠されがちな工程ではあります。しかし、厳密なソフトウェア品質が求められる自動車、医療、航空の分野では、ISO 等の国際規格で単体テストを実施することがルール化されており、このことがソフトウェアの品質安定に寄与しています。

そして、単体テストはソフトウェアの開発工程において唯一、実装されたソフトウェアの全数試験（全数テスト）が行える工程であり、また最初のテスト工程でもあります。もちろん、文字どおり"全数"のテストというのは開発工数上、現実的な話ではありませんが、単体テスト自体を避けていては、本当に高品質なソフトウェアの実現は望めません。

ここからは、単体テスト（Unit Test）の手法について説明します。単体テストには大きく分けて、以下の 2 つの手法があります。

- ブラックボックステスト：

  ソフトウェアの内部構造を参照せず、仕様書に基づきテストケースを設計する手法

- ホワイトボックステスト：

  ソフトウェアの内部構造に着目し、テストケースを設計する手法

簡単にいえば、仕様書どおりに関数が実装されているかを確認することがブラックボックステストであり、関数の内部構造に問題がないかを確認することがホワイトボックステストです。

5.1 ブラックボックステストとは

5.1.1 概要

ブラックボックステストは、ソフトウェアの内部構造を参照せず、仕様書に基づきテストケースを設計するテスト手法です。

「仕様書に基づきテストケースを設計する」という考え方はとてもシンプルで、機能仕様を関数仕様書に落とし込み、関数テスト設計書を作ってテストを行うことが望ましく、そうすることで効果的なブラックボックステストを行うことができます。

ブラックボックステストでは、仕様上定義された入力値を入れ、関数を実行した結果（出力値）が期待どおりであるか、つまり、仕様どおりであるかを確認するのですが、関数の入出力だけをテストすればよいかというと、そうではありません。以下に、ブラックボックステストで使われるテスト技法のなかでも、代表的なものをいくつかピックアップしてご紹介します。

5.1.2 同値分割手法

同値分割は、予想される出力結果をグループ分けし、そのグループ内の代表値を入力して正しい出力結果になるかを確認するテスト手法です。ソフトウェアが正常に動作する値を有効同値クラス、エラーになる値を無効同値クラスとして分類し、その中で代表値を決めてテスト条件を決定していきます。

代表値のテスト結果が、そのままクラス内のすべての値に反映されるという考え方がベースとなっており、全数試験よりも少ないテストケースで広い対象範囲を網羅することができ、効率的なテスト手法であるといえます。以下に、

そのテスト手法例を挙げます。

たとえば、年齢入力に関する機能を実装する際、仕様で 0 歳～100 歳までを入力可能と定義している場合は、下記のように同値クラスの分割を行い、その中から任意の代表値を選んでテストケースを作成します。

> 同値クラスの分割

有効同値クラス：0 歳～100 歳

無効同値クラス：-1 歳以下、101 歳以上

> 代表値の選択とテストケース作成

テスト 1：テストデータ＝50（有効同値クラス）

テスト 2：テストデータ＝-10（無効同値クラス）

テスト 3：テストデータ＝-110（無効同値クラス）

代表値を増やすことは問題ありませんが、テスト項目を増やせば増やすほど全数試験に近づいてしまい、意味のないテストデータが増えてしまいます。そのため、その機能の特性に合わせて有効なテスト設計を行うことが重要となります。

5.1.3 境界値分析手法

境界値分析は、同値分割で作成した同値クラスの境界に当たる数値を入力し、正しい出力結果になるかを確認するテスト手法です。有効同値クラスの最大値と最小値が境界値となり、境界値と、境界値の 1 つ下の値、境界値の 1 つ上の値でテストを行います。

境界値については、ソースコードの中でソフトウェア開発者が仕様書の内容を誤解していたり、コーディングを誤っていたりと、不具合が発生しやすい箇所であるため、境界値分析は不具合の発見に効果的なテストであるといえます。

ここで、5.1.2 で使用した年齢入力に関する機能を実装する例を考えてみましょう。境界値分析では、以下の値をテストデータとしたテストケースを作成します。

境界値：0、100

境界値±1：-1、1、99、101

これに加えて、同値分割で選んだ代表値のテストも行うことで、より堅牢なテストを行うことができます。

5.2 ホワイトボックステストとは

5.2.1 概要

ホワイトボックステストは、ソフトウェアの内部構造に着目し、テストケースを設計するテスト手法です。

ホワイトボックステストでは、ソフトウェアの内部構造を網羅するようなテストケースを作成し、作成したテストケースがプログラムをどのくらい実行（網羅）できているかのテスト品質指標としてカバレッジ（網羅率）を計測します。ブラックボックステストと違い、関数の仕様を理解していなくてもカバレッジを 100%にすることが可能なため、比較的簡単に取り組むことができるテストといえます。

時折、「自社ではカバレッジを100%にしているので何も問題ない」というお客様の声を耳にすることがありますが、カバレッジ 100%＝不具合 0 ということではないため、この点には注意が必要です。カバレッジが 100%というのは、あくまでもテスト対象関数のすべての命令を実行した、という事実でしかないからです。カバレッジの網羅は、実行されないデッドコードの発見や永久ループの発見にはつながりますが、たとえカバレッジが 100%であっても、それが仕様どおりであって不具合がないと判断できるかといえば、それは全く別の話となります。そのため、ブラックボックステストで仕様どおりに実装できていることを確認し、かつ、ホワイトボックステストでカバレッジを計測しつつ、その過程で不具合を発見し、プログラムの堅牢性を確認するといった両立が必要となります。

5.2.2 カバレッジ（網羅性）について

5.2.2.1 概要

ホワイトボックステストでは、主にソースコードの命令文や分岐条件の実行有無を計測しますが、特にこのソースコードがテストされた割合のことをコードカバレッジと呼びます。コードカバレッジにはいくつかの基準があります。

より厳格なカバレッジ基準を目指せば、その分、テスト設計に必要な工数も増えていくため、そのプログラムがシステムに与える影響度合いによって適切なカバレッジ基準を選び、テストを実施することが重要となります。以下に、代表的なコードカバレッジ基準をいくつかご紹介します。

5.2.2.2 ステートメントカバレッジ（命令網羅／C0）

ステートメントカバレッジは、日本語で「命令網羅」と呼ばれるように、ソースコードの実行可能な命令文全体のうち、テストでどのくらい実行したかを評価します。

ステートメントカバレッジでは、条件分岐があった場合、いずれかの条件のパスを通って命令が 1 回実行されると、その時点でカバレッジが達成されたとみなすため、コードの網羅性が低いカバレッジ基準であるといえます。

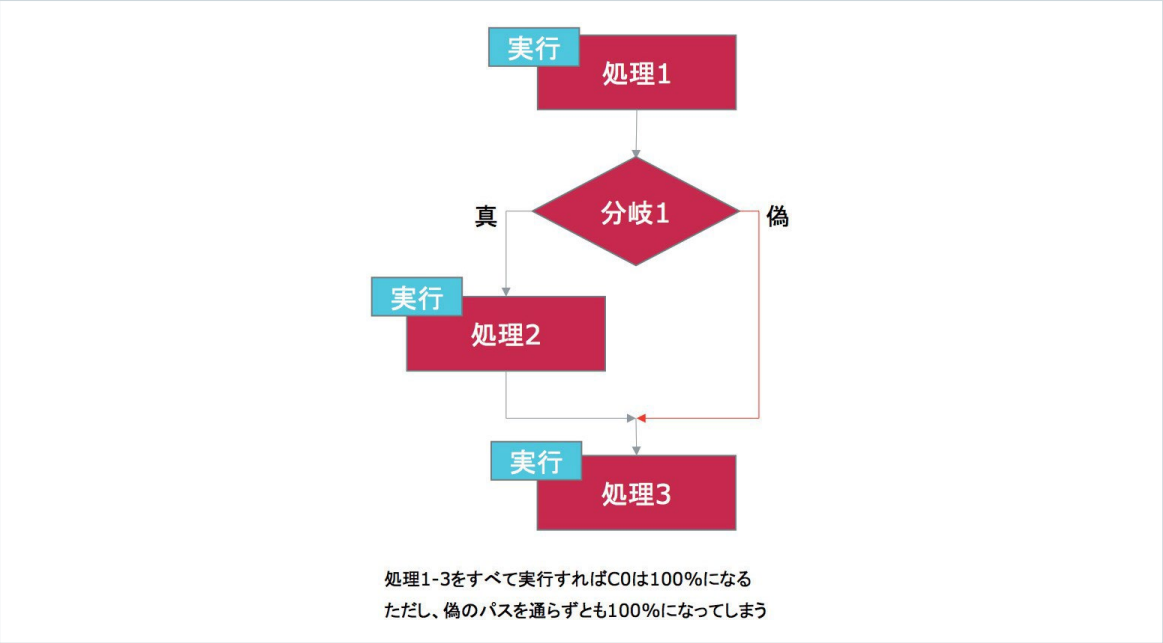


図 9: ステートメントカバレッジ

5.2.2.3 ブランチカバレッジ(分岐網羅／C1)

ブランチカバレッジは、日本語で「分岐網羅」と呼ばれるように、ソースコードの判定条件(if 文等)の真偽を、テストでどのくらい実行したかを評価します。

ブランチカバレッジでは、真偽の両方を評価するため、ステートメントカバレッジよりも網羅性の高いカバレッジ基準であるといえます。ただし、ブランチカバレッジでは、判定条件内の AND や OR を含む複合条件の組み合わせまでは考慮しないため、すべての条件の組み合わせを網羅できるわけではありません。ブランチカバレッジが 100%網羅されているのであれば、同時にステートメントカバレッジも 100%を達成できます。

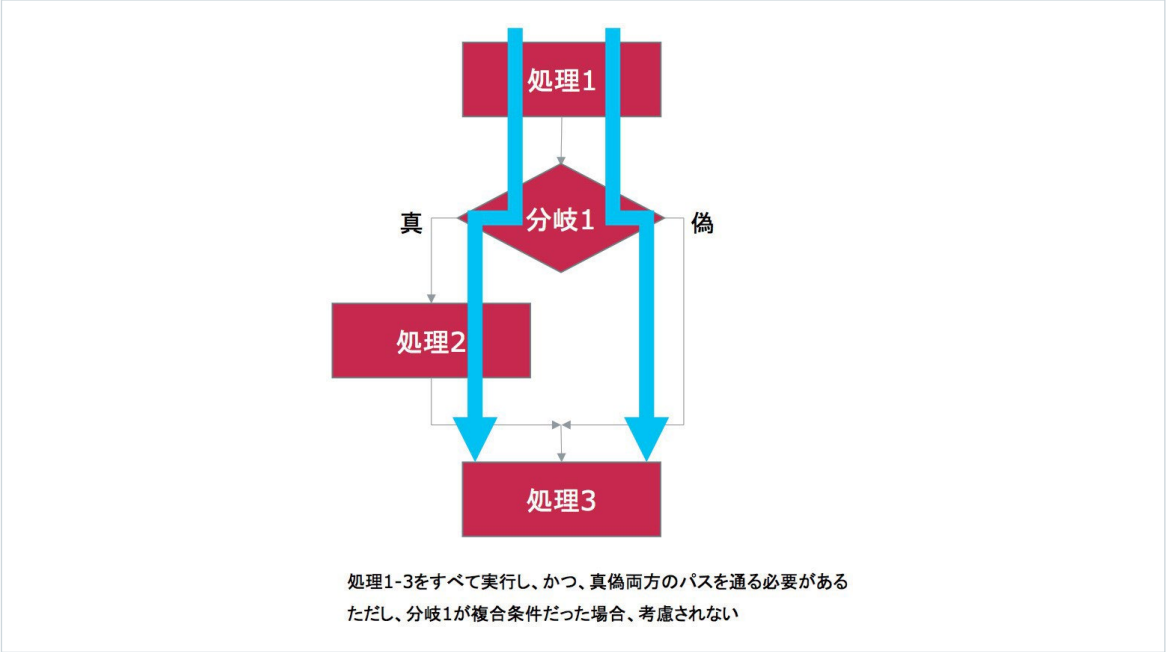


図 10: ブランチカバレッジ

5.2.2.4 コンディションカバレッジ（条件網羅／C2）

コンディションカバレッジは、カバレッジ基準の中で最も厳格なコードカバレッジです。テスト対象の条件分岐に複合条件が使われる場合でも、各条件の真偽すべての組み合わせをテストします。ソースコードに対する網羅性が高い一方で、テストケース数が膨大になるため、規模の大きなソフトウェアでコンディションカバレッジまで網羅されているケースは少ないといえます。

5.2.2.5 MC/DC カバレッジ（Modified Condition/Decision Coverage）

MC/DC カバレッジは、「Modified Condition/Decision Coverage」の略です。

コンディションカバレッジと同様にコードの網羅性が非常に高いため、航空機向けソフトウェアや自動車に搭載するソフトウェアなど、高い安全性が求められるケースにおいては MC/DC カバレッジ基準を用います。複数あるカバレッジ基準の中で、現実的かつ最も厳しい基準が MC/DC であるといえるでしょう。

MC/DC カバレッジの定義は下記のとおりです。

【MC/DC カバレッジの定義 [5]】

MC/DC requires all of the below during testing

1. Each entry and exit point is invoked.
2. Each decision takes every possible outcome.
3. Each condition in a decision takes every possible outcome.
4. Each condition in a decision is shown to independently affect the outcome of the decision.

抄訳になりますが、MC/DC カバレッジはテストで以下の要件を確認することを求めています。

[5] 「A Practical Tutorial on Modified Condition/Decision Coverage」; NASA / TM-2001- 210876; May 2001; https://shemesh.larc.nasa.gov/fm/papers/Hayhurst-2001-tm210876-MCDC.pdf

1. プログラムのすべての入口と出口を通過していること
2. プログラムの各判定式が、すべての取りうる結果を出力すること
3. プログラムの判定式の各条件が、すべての取りうる結果を出力すること
4. プログラムの判定式の各条件が、判定結果に単独で影響すること

たとえば、とある複合条件式がある命令において「A and B が真の時、処理 2 を実行する」場合、A が真であれば、B が真偽のどちらかによって判定の出力が変化しますが、A が偽であれば、B の真偽がどちらでも判定は偽になります。つまり、A が偽のテストケースは 1 つでよいため、このソースコードにおける MC/DC のカバレッジを 100%にするためのテストケースは 3 通りということになります。

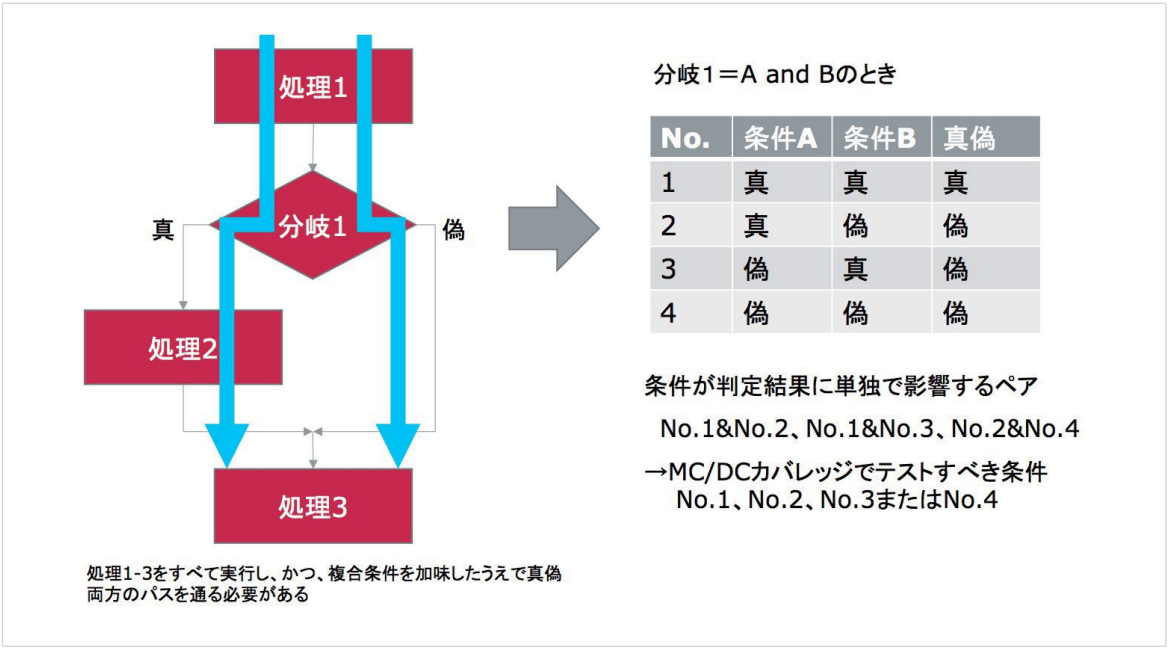


図 11: MC/DC カバレッジ

6. まとめ

手作業での単体テストには莫大な工数が発生します。ブラックボックステスト、ホワイトボックステスト、それ以外のテストをどのような基準で行うのか。各製品ドメインに求められる品質基準を考慮しながらテスト計画を定め、きちんと運用していくことが重要です。

また、手作業でのテストにはヒューマンエラーが介在しますので、各種国際認証におけるテストエビデンスにはなり得ません。そのため、何かしらの自動化ツールを導入してヒューマンエラーを排除し、均一的なテストエビデンスを作成できるような環境を整えることが重要です。

ベクターが開発、販売している単体テスト自動化ツール「VectorCAST（ベクターキャスト）」は、そうした環境構築に役立つ実績あるツールとして世界中の開発者の皆様にご使用いただいています。今回は、単体テストについての基本的な知識を解説させていただくに留めますが、「VectorCAST」について解説した「はじめてのVectorCAST」も機会があれば是非、ご一読いただければと思います。

「VectorCAST」については、弊社 Web サイトでもご紹介しております。ツールに関する相談、お問い合わせは、お気軽にベクター・ジャパンまでお寄せください。

> ベクター・ジャパン Web サイトはこちら
> **www.vector.com/jp/ja/**
> VectorCAST 紹介ページ「世界で選ばれる単体テストツール VectorCAST」はこちら
> **www.vector.com/jp/ja/products/products-a-z/software/vectorcast/vj-unit-testing-solution/**
> VectorCAST に関するお問い合わせ・資料希望はこちら
> **www.vector.com/jp/ja/products/products-a-z/software/vectorcast/inquiry-vectorcast/**

以上、本稿が皆様の開発業務の一助となれば幸いです。

7. お問い合わせ窓口

技術関連のお問い合わせは、ベクターサポートまでお寄せください。

✓ サポートリクエスト（Web サイト経由のサポート問い合わせフォーム）
www.vector.com/jp/ja/support-downloads/support/

【お問い合わせ時のお願い】

お問い合わせの際、サポート開始前に製品のバージョン（例：CANape バージョン xx.x など）、シリアルナンバーなどをお伺いいたします。お手数ではございますが、事前にご確認のうえ、お問い合わせくださいますようお願い申し上げます。

【サポート対応について】

夜間／休日に頂いたお問い合わせは翌営業日の対応となります。また、内容によりご回答までにお時間を頂く場合がございます。なお、不具合が発生した際にご使用されていた関連プログラム一式のコピーを、検証のためご提供いただくようお願いする場合がございます。何卒ご理解ご協力のほど、よろしくお願い申し上げます。

✓ **お見積もり／保守契約関連のお問い合わせ**

営業部　TEL：（東京）03-4586-1808　E-mail：sales@jp.vector.com

VECTOR >

MORE INFORMATION
Visit our website for:

> News
> Products
> Demo software
> Support
> Training and workshops
> Contact addresses

vector.com