72

組込みシステムのユニットテスト効率化のための スタブ自動生成と仕様検証技術[1]

1. 概要

本編では、電子楽器システム開発におけるユニットテストのスタブ自動生成技術と仕様検証の方法について、ヤマハ株式会社（以下、同社とする）の事例を紹介する。「同社のスタブ自動生成の技術解説と、仕様検証のためのテスト因子抽出法の二つを説明したのち、その効果についても報告する。

組込みシステム製品における近年の大規模化・複雑化は、多くのメーカーの課題であるが、同社主力製品である電子楽器もその例に漏れず開発上の課題となっている。組込みシステム製品に求められる高い品質に対し、検証に必要なコストは上昇の一途を辿っており、検証方法の効率化が急務となっている。

同社では DI コンテナ技術を用いたソフトウェアフレームワークを開発し、電子楽器製品への適用を行った。DI コンテナ技術の適用については、別事例「組込システムのモデルベース開発適用における DI コンテナの活用」[2]を参照していただきたい。検証工程に DI コンテナ技術を応用し、クロス開発環境やスタブ・モックを効率的に作成できるテストフレームワークを実現した。また、モデルベース開発適用の課題の一つであるモデル検証の方法として、上流工程におけるテスト因子の抽出方法に工夫を加えたことで、品質向上を果たすことができている。

2. 取り組みの目的

2.1. 適用前の電子楽器開発の状況

同社の電子楽器は 1990 年代以降のネットワーク化への対応を経て、急激な進化を遂げた。しかし、それを支えるソフトウェアシステムの大幅な見直しが行われず、長年にわたり流用を重ねざるを得なかった。その結果、2009 年頃にはソフトウェアの複雑化が進行し、品質低下が顕著となった。2009 年時点での同社電子楽器製品を調査した結果、ソースコードベースでの流用率は約 95.9%と高い流用率にもかかわらず、品質確保に必要なソフトウェアテストの工数が上昇し、開発全体を圧迫する状況に陥っていた。さらに減少傾向だったバグ密度も増加に転じていた（図 72-1）。

[1] 事例提供: ヤマハ株式会社 楽器・音響開発本部 安立 直之 氏

図 72-1　流用率とテスト工数、バグ密度

特にバグ密度が増加に転じた 2008 年と 2009 年に着目し、開発メトリクスを精査した。その結果、実装までの上流工程の工数が減少しているのに、システムテストを中心とした後工程の割合が急増したことが判明した（図 72-2）。これは一つには複雑化したシステムの分析に時間が掛かっていること、もう一つはシステムテストが結果的にビッグバンテスト化していると推測される。当時の開発担当者から開発プロセスの問題点として、複雑化したソフトウェアの分析に多大な時間が掛かること、システムテストで不具合が多発し、その原因究明に時間が掛かっていることが挙げられている。

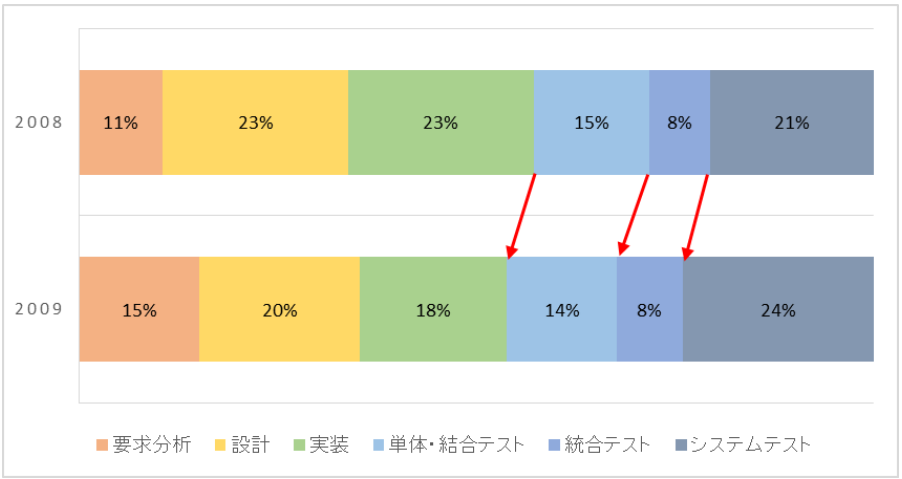


図 72-2　検証工程毎の比率推移

こうした状況を鑑み、同社ではソフトウェア開発の改善を模索した。そこで辿り着いた結論は、フレデリック・ブルックスの「銀の弾丸などない」[7]という言葉であった。ある設計手法、品質技術で解決するということはない。小さな改善の積み重ねが大きな改革に繋がる。そのためには従来のシステム設計の範囲に留まることなく、ソフトウェア開発全体に潜む問題を一つ一つ分解し改善しなければ期待する効果は得られない。まず、課題を大きく５つに分類し、課題の解決策と具体的手法をまとめた。それが図 72-3 のソフトウェア開発プロセス改善ビジョンである。改善がどこに向かうかビジョンを示し、部門長の理解を得られたことで、本事例となる取り組みが開始されるに至った。

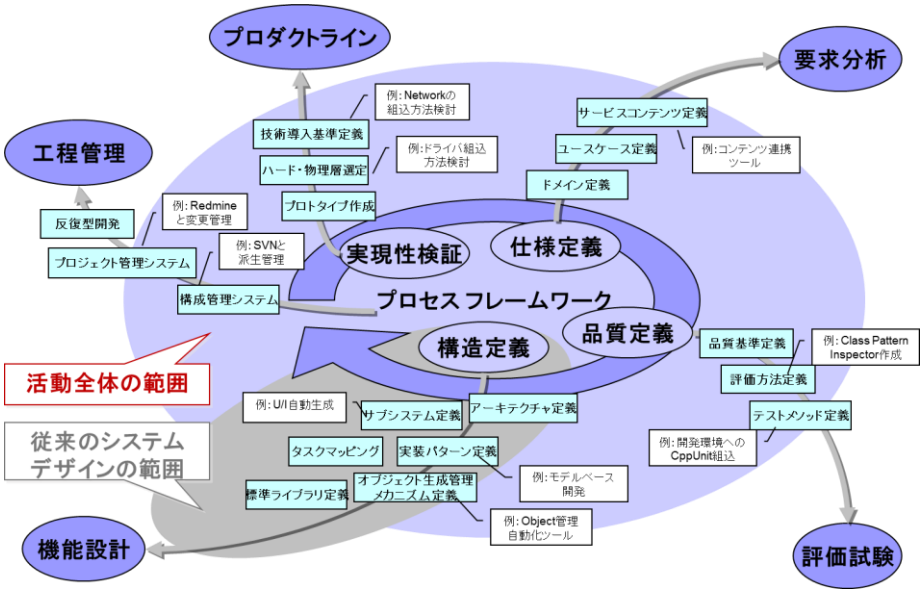


図 72-3　ソフトウェア開発プロセス改革ビジョンの提案

2.2. 課題と目標

同社では、「組込システムのモデルベース開発適用における DI コンテナの活用」[2]の通り、DI コンテナ技術の応用によりモデルベース開発のコンポーネント化を推進している。そして、同時に改善すべき「2.1」で明らかになった品質面での２点の課題がある。

課題１：検証工程でのビッグバン結合を防止するため、コンポーネント単位で品質確保を実現すること。

課題２：モデル化されたソフトウェアを上流工程で検証し、要求分析の効率化を図ること。

最終的に達成すべき目標は、同社のソフトウェア生産性向上により新機能開発を容易にし、商品価値を高めることにある。そのために２点の課題を解決した上で、適用前の問題であるバグ指摘率の低減と、品質確保にかかる工数の削減を本取り組みの目標とする。

3. 取り組みの対象、適用技術・手法、評価・計測

取り組み対象は、冒頭で述べたとおり電子楽器本体の組込みシステムである。DI コンテナを適用技術の主体とし、その技術を応用したテスト手法を展開する。「2.2」において２点の課題を取り上げたが、開発プロセス上での本取り組みの対象範囲を図 72-4 に示す。課題１のコンポーネント単位での品質確保は、DI コンテナ技術を応用したテストフレームーワークの利用によって解決を図る。課題２のモデル化されたソフトウェアの検証は、テスト因子抽出と仕様検証によって解決を図る。

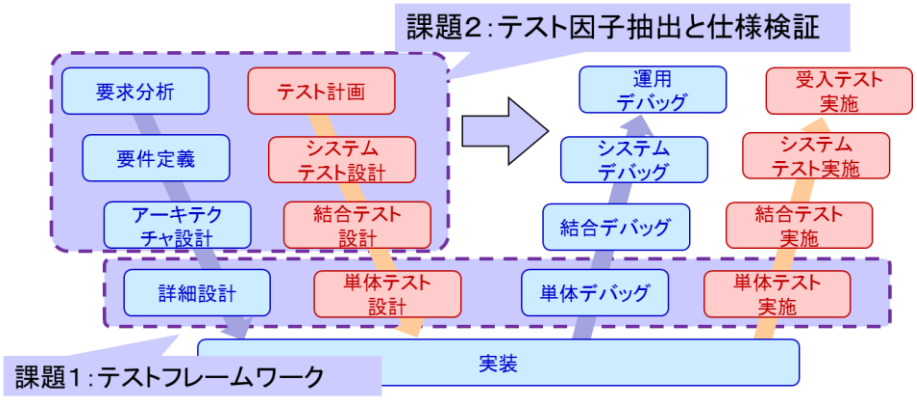


図 72-4　W字モデルを元にした本取り組みの対象範囲

それぞれの課題において、具体的に次の技術を用いる。

課題１の解決方法：テストフレームワーク

- 同社独自開発の組込みシステム向けフレームワーク「CoPaN」
- 自動 Stub 生成技術

課題２の解決方法：テスト因子抽出と仕様検証

- テスト駆動開発
- MECE 仕様検証法
- デシジョンテーブル

個々の技術については「4. 取り組みの実施、及び実証上の問題・工夫」以降の取り組みの実施において解説する。システム改善の評価として、開発工数・バグ密度などプロジェクトメトリクスによる品質測定を行う。

4. 取り組みの実施、及び実証上の問題・工夫

4.1. DI コンテナを用いたフレームワーク「CoPaN」とユニットテストの効率化

4.1.1. OS 依存性の内包とクロス開発環境の実現

CoPaN は C++言語をベースに開発したもので、Embedded C++の規約に準拠し実装された軽量フレームワークである。CoPaN は DI コンテナとしてのインターフェースとコンテナを併せ持ち、実体となるオブジェクトの基本クラスとしての機能を有する。CoPaN フレームワークは DI コンテナとして関連を抽象化するだけでなはなく、関連に必要なコンポーネント間通信も内包している。全ての関連は依存性定義で帳票管理されているが、関係を結ぶ際の通信方法も定義されており、通信方法毎に内部実装を継承した Interface クラスが選択され、実行時に定義された通信方法と実行コンテキストでインターフェースの呼び出しが行われる（図 72-5）。

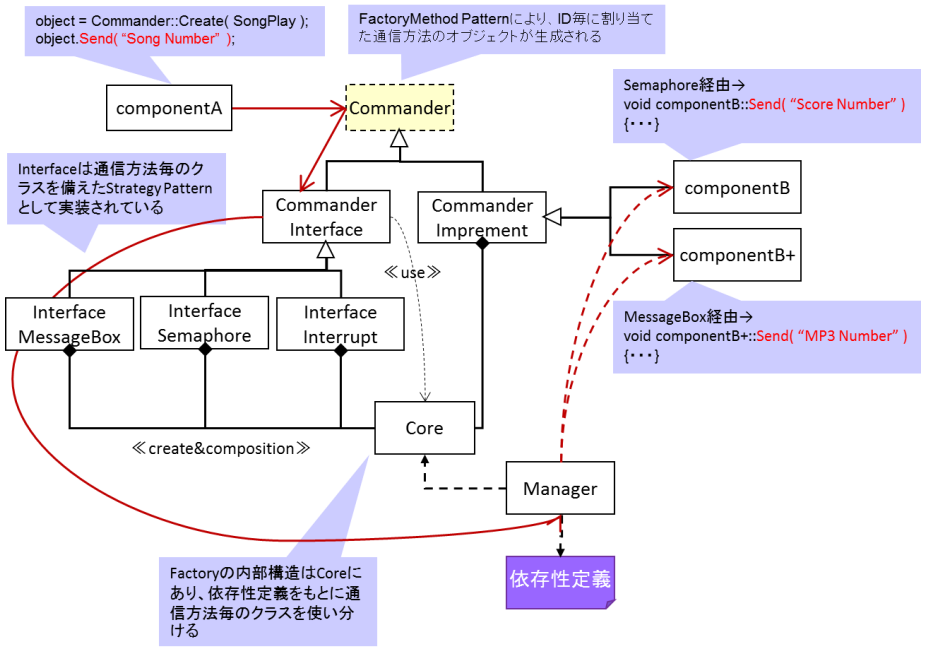


図 72-5　OS 依存性の内包と通信方法変更の仕組み

クロス開発環境の実現においても、この技術を応用する。前述の通り CoPaN フレームワークにはターゲットデバイスのシステムコールに対応した Interface クラスが定義されているが、新たにホストコンピュータのシステムコールに対応した Interface クラスを追加することで、ソフトウェアシステムをそのままホストコンピュータ上で動作させることが可能になる（図 72-6）。例えば Windows PC 上でクロス開発環境を作りたい場合は、Interface クラスを継承し Windows のスレッド間通信を行うクラス「InterfaceWinMeesage」を追加する。次に依存性定義を書き換え、追加した「InterfaceWinMessage」を使用するように修正する。この方法によって同社では Visual Studio 上にユニットテスト環境を構築し、Windows 上で全てのプログラムの実行、テストが可能となった。

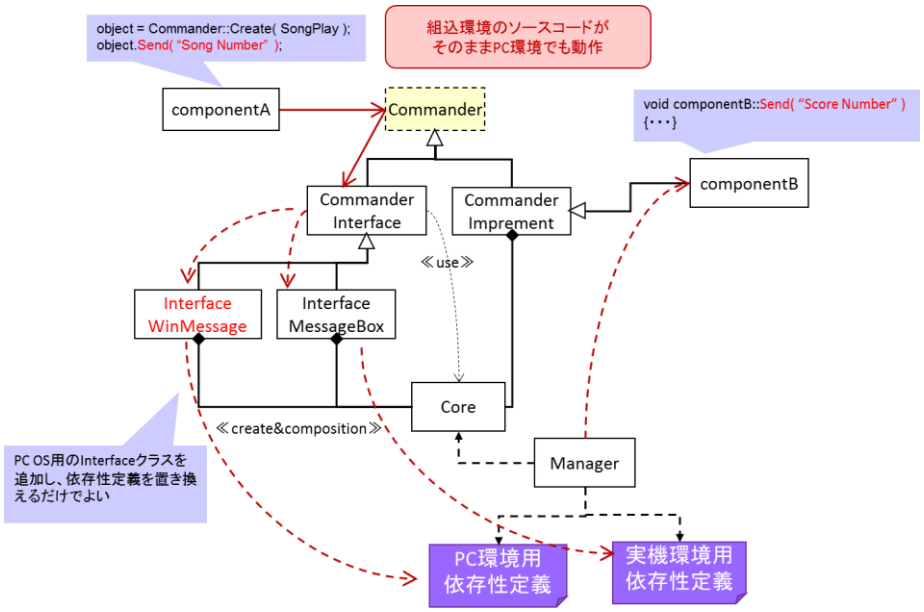


図 72-6　クロス開発環境の実現方法

4.1.2. フレームワーク「CoPaN」を用いたスタブ・モックの生成と置換

「4.1.1」で述べたとおり、容易にクロス開発環境を構築し、ホストコンピュータ上でユニットテストを実施することが可能となったが、実際にユニットテストを行う際に大きな障壁となるのが、スタブ・モックの生成と、テスト環境における置き換えである。ここでシステム上の１つのクラスに対してユニットテストを行うケースについて考えてみる。componentA は依存先として componentB、componentC、componentD の３つのクラスがあるとする。componentA をテストする場合、componentB、componentC、componentD をテスト用のプログラムに置き換え、componentA からの出力結果が正しいかをテストしなければならない。そのため３つのクラスに置き換わる Stub をプログラミングし、コンパイル対象として componentB、componentC、componentD を取り除いて stubB,stubC,stubD に変更する作業が必要となる。また、状態遷移などの動的な振る舞いを確認するため、componentB の動作を模した mockB に置き換える必要がある場合もある。これをシステム上の全てのクラスについて行うと、全クラス数×ｎ個の依存関係のスタブ・モックを実装し、コンパイル対象の置き換えをしなければならない。

スタブ・モック実装の工数削減のため、同社ではフレームワーク CoPaN を用い、標準 Stub クラスによるスタブ生成の自動化と DI コンテナを用いた動的置換という方法で解決を図った。の通り、標準 Stub クラスは、DI コンテナの受け側に当たる Implement クラスを継承して実装し、インターフェース毎の呼び出し回数と引数の値を保存する機構を有している。そのため、いつ・どのインターフェースが・どのような値で呼び出されたかを検査することができる。この機構を用いることで、ユニットテストのテストコードで自由に任意のコンポーネントをスタブに置き換え、実行結果を検査することを可能としている。図 72-7 では componentA の呼び出し先である componentB を標準 Stub クラスに置き換え、componentA の実行結果を検査する例を示

している。ユニットテストのプログラムには、最初に標準 Stub クラスのインスタンスを宣言する。このとき標準 Stub クラスには componentB に関連づけられている ID='PlaySong'を与える。標準 Stub クラスのコンストラクタ内部で依存性定義ファイルを書き換え、その時点で componentB への関連付けは抹消され、ID='PlaySong'に関連付けられるのは標準 Stub クラスのインスタンスになる。次に、componentA に対するテストを実行すると、その結果、本来 ID='SongPlay'で componentB が呼び出される部分で標準 Stub クラスが呼び出され、呼び出された時の値が格納される。標準 Stub クラスから ID='PlaySong'の結果を取得し、実行結果を検査することでユニットテストが完了する。さらに最初に宣言した標準 Stub クラスのインスタンスを破棄すると、自動的に依存性定義を元に戻すという仕組みを備えている。

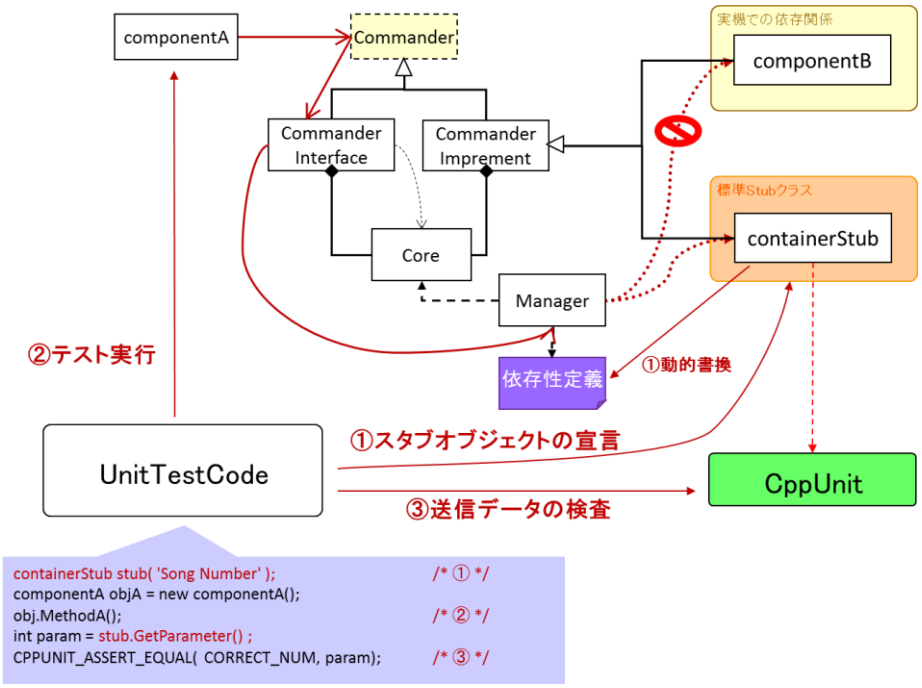


図 72-7　標準 Stub の利用方法

同様にモックの生成も可能である。標準 Stub クラスを継承し、テスト用のモックを作成することもできる。その際テストで必要なインターフェースのみをモックに実装し、それ以外のインターフェースは標準 Stub クラスと同じ入力の記録を行わせることも可能である。ユニットテストにおけるモックの使用方法は標準 Stub クラスと全く同じ方法になる（図 72-8）。

以上の通り、クラス毎に Stub クラスの実装工数削減を実現した。さらにテストコードの実行時にスタブへの置換が可能なため、テスト内容に応じて自由な順序でスタブ化し多様なテストにも対応が可能となった。また、テストコードのカバレッジ計測の一つとして、テスト対象のクラスからの全ての依存関係が検査されているかを記録、確認することも可能としている。

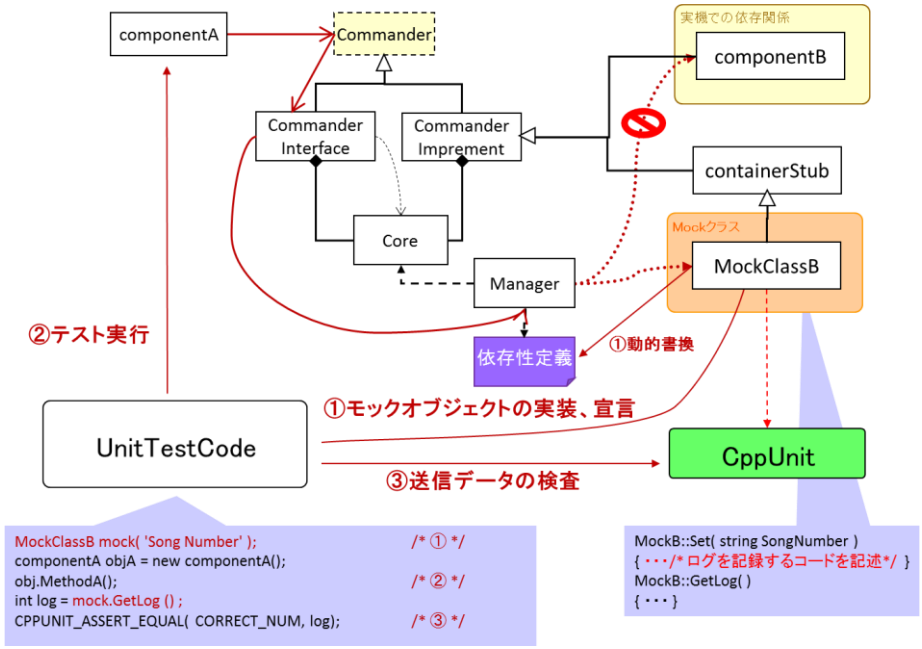


図 72-8　Mock クラスの実装と使用方法

4.2. テスト因子抽出による仕様検証法

4.2.1. 上流工程の課題

ソフトウェア開発プロセスの上流工程における活動は、要求分析・要件定義から成る。上流工程での課題を品質視点で調査したところ、「設計者自身による設計品質作り込みーテストエンジニア視点の活かし方ー」によれば「仕様記載漏れが 53%」という解析結果がある[5]。要求仕様書の品質特性（IEEE 830-1998）で定義される品質特性のうち「正当性」および「完全性」が損なわれた場合、リスクが高くなるとし、中でも「完全性」を重要問題として取り上げている[5]。同社はここに着目し、「完全性」の担保について考察した。ロジカルシンキングにおける完全性を追求する概念として「MECE(Mutually Exclusive and Collectively Exhaustive)」[2]がある。要求仕様書が MECE であることを検証することができれば、完全性の確保に繋がると推測した。しかし要求仕様書という自然言語について MECE 性を検証することは、ほぼ不可能である。

そこでもう一つの観点として、テスト設計における漏れの原因を考察した。一般的な組み合わせテストの設計プロセスは、機能分析、因子・水準の抽出、そして直交表やデシジョンテーブルを用いた組み合わせ設計、という流れで行われる。個々のプロセスに漏れが生じることがテストの欠陥に繋がる。裏を返せば要求仕様書の記述段階で、テスト設計に必要な分析が行われ、その記載があれば、漏れを防止できると考えられる（表 72-1）。

[2] 「相互に排他的な項目による完全な全体集合」の意。コンサルタントファーム・マッキンゼー社が問題解決の方法として考案した。

表 72-1　テスト設計プロセスと要求仕様書の対比

テスト設計プロセス	要求仕様書の記載
機能分析	仕様書の大項目に相当
因子抽出	仕様書の小項目や動作仕様の内容に相当
水準抽出	動作の有効範囲を示す水準は記載されるが、他は仕様書には記載されることはない
組み合わせ設計	特異なケースを除き仕様書に記載されることはない

4.2.2. 因子の MECE 性に注力した仕様検証法

考察を元に、同社では要求分析段階において要求仕様書の機能及び因子の漏れがないことを検証することで、完全性を担保する方法を開発した。

(1)　要求仕様書を作成する
(2)　マインドマップを用いて要求仕様書の文章から因子抽出を行う
(3)　抽出された因子について、因子間の漏れ・ダブりがないか MECE 性をレビューする
(4)　レビュー結果を要求仕様書にフィードバックし、要求分析の漏れを防ぐ

この方法で鍵となるのは（2）の因子抽出の方法である。テストケース作成を前提とした因子抽出では、テストの粒度や境界値の考慮など、様々なケースを想定する必要があるが、ここでは要求仕様書の完全性のみを確認する事に注力するため作業化を計った。図 72-9 に示す通り、まず要求仕様を一文ごとに分割し、マインドマップに写す。全て写した段階で文章の内容をさらに分割し、因子となり得る「条件」、「トリガー」と「結果」に相当する単語に整理する。従って整理されたマインドマップには要求仕様書に記載されている範囲の因子のみが抽出された状態になる。ここでマインドマップのツリーに偏りがないか、因子の漏れやダブりがないか、因子の粒度が同じか、MECE 性を検証する。検証の結果、曖昧な仕様や考慮不足があれば、因子を追加・修正する。この方法を採用する以前は要求仕様書をレビューする際、第三者の書いた文章そのものの読解に非常に時間が掛かっていたが、マインドマップを用いることで第三者への視認性が向上し、因子の関係性のみをレビューすることに注力できるため、レビュー工数の削減と問題発見率の向上に繋がった。

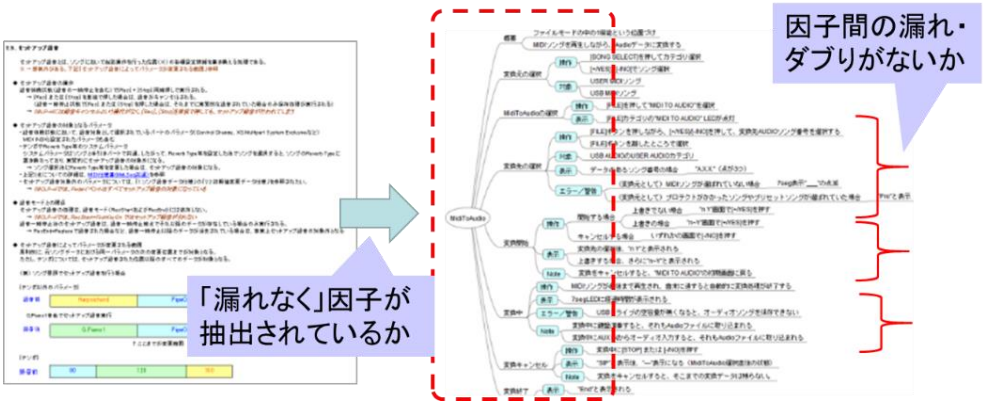


図 72-9　MECE 仕様検証法

4.2.3. 仕様分析とテストケース導出

「4.2.2」で作成したマインドマップは、次工程のテストケース作成に用いられる。既に因子となる「条件」「トリガー」と「結果」が抽出されているため、デシジョンテーブルの項目に写すことで行方向が網羅できる。従ってテストケース作成では組み合わせの考慮に注力できる。また、設計・実装過程で新たな仕様要件や制約や見つかった場合、デシジョンテーブルに行列を追加するだけで対応が可能となり、テストケース作成の手戻りを最小に抑えることにも繋がる。（図 72-10）

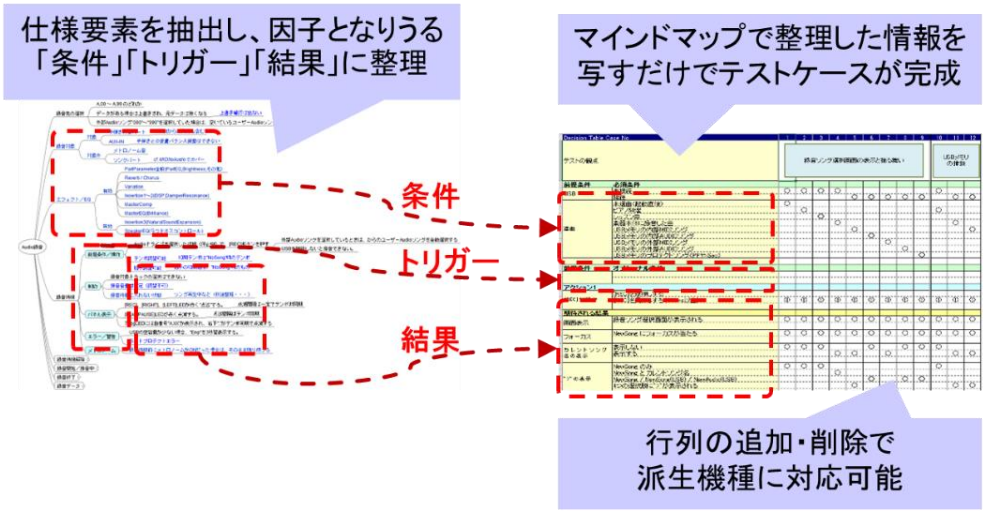


図 72-10　仕様分析とテストケース作成

4.3. 品質維持のためのプロジェクトマネジメント

「4.1」「4.2」で見てきた通り、同社では上流工程・下流工程への２つのアプローチを実施したが、大規模開発における運用方法についてプロジェクトマネジメント観点から解説する。

品質を証明するには、全ての製造物に対する検証結果が適切かを示す必要がある。しかしそれを証明するために、要求仕様書の全ての文章、単語に対応するテストケースを照合することは不可能であり、ソースコードの全ての行についてユニットテストが存在するか照合することも難しい。ソースコードについてはカバレッジ計測などの方法が存在するが、実際には限られた資

源・時間の中でカバレッジ率を 100%にすることは困難である。また、同社はソフトウェアテストのための専門部署が存在せず、開発者が自らテスト設計とテストの両方を行う環境にある。そうした限られたリソースの中での現実的なアプローチについて、同社が採用した事例を解説する。

4.3.1. テストファースト開発

ここまで解説した２つの技術は、単体テストと、結合・統合テストの効率化のための手法である。実際に設計・実装と単体テスト、結合・統合テストとがどのような関係で実施されるかだが、同社では２つのプロセスについてテストファースト開発を導入した。一般的なテスト駆動開発は、最初にテストコードを記述し、そのテストが動作する実装を行う手法である。同社は実装におけるテスト駆動開発に加え、仕様分析及び結合テスト・統合テストにもテストファーストを適用することで、上流工程からの品質向上を図った。4.2.で解説したとおり仕様分析時に因子抽出を行い、設計・実装前に結合テストケースを作成するフローである。単体テストレベルと結合・統合テストレベルの２段階にプロセスを絞り込み、軽量な反復型開発を実現した（図 72-11）。

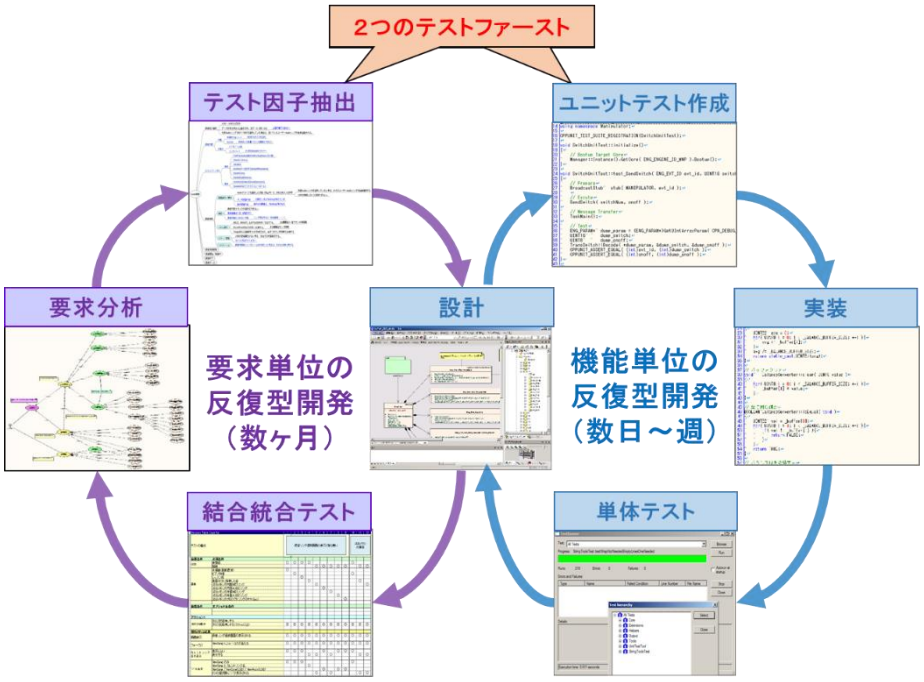


図 72-11　２段階テストファーストプロセス

4.3.2. テストファースト開発のマネジメント

ソフトウェア品質保証で考慮すべき観点として、テストカバレッジがある。先ほど述べたとおり、ソースコード一行一行のカバレッジを確保することは現実的には困難である。そこで同社では、テストファースト開発が全ての実装において実施されているかを、成果物の時系列だけで判断する方針を採用した。テストファースト開発が行われていれば、開発の各過程で生じる成果物も適切な順序で出力される筈である。従って成果物の順序がテストファースト開発のフロー通りかを確認すれば、マクロ視点では順調に品質保証に必要な活動が行われていると判断でき

る。同社では図 72-12 のテストファースト開発の基本フローを定めると同時に、成果物の中でも特に重要なソースコード、ユニットテスト、結合テストケース、結合テスト結果の４点に対し、リポジトリへのコミット時間を抽出し確認することにした。４点の成果物が以下の条件に合致していれば、少なくとも実装以前にユニットテスト・結合テストのテストケースが作成されていると推定されるからである。

・ソースコードとユニットテストコードが同時にコミットされているか
・ソースコードの最終コミット以前に結合テストケースがコミットされているか
・ソースコードの最終コミット後、結合テスト結果がコミットされているか

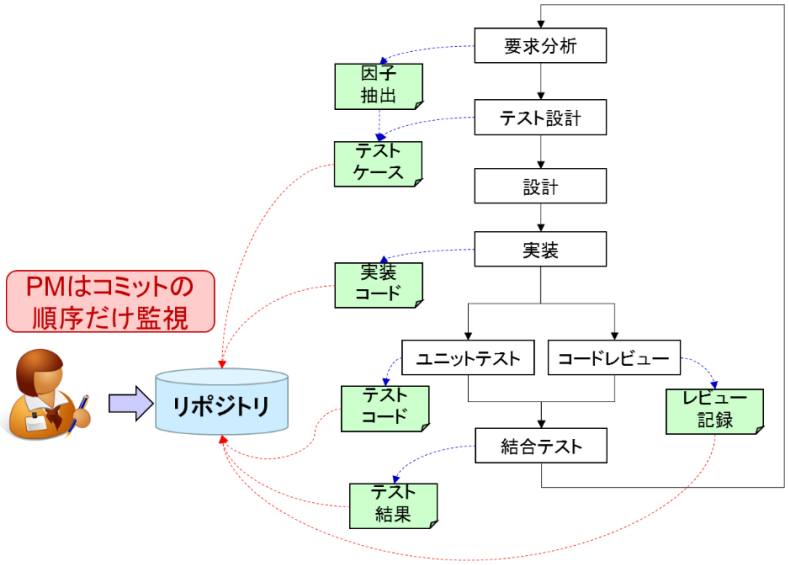


図 72-12　テストファースト監視のためのフロー

同社では要求分析から検証に至る全ての成果物をリポジトリ管理し、Jenkins[3]を用いて継続的インテグレーションを実施している。日々の定期ビルドに、リポジトリのコミットログから成果物の状況を自動集計するシステムを開発した。その際に課題となるのは、個々の成果物のトレーサビリティである。そこで同社では２つのルールを設けることで解決を図った。ソースコードとユニットテストコードとの紐付けは、両者を同時にコミットするプロセスルールを設け、コミットログから抽出可能とした。ソースコードと結合テストケースの紐付けは、コミット時のコメントに機能名をタグとして記述することで実現した。このルールは開発者に新たな作業負荷を掛けることなく、作業手順を明確にするだけで自動集計が可能となり、品質状況の可視化を可能とした（図 72-13）。

[3] Jenkins オープンソース継続的インテグレーションツール

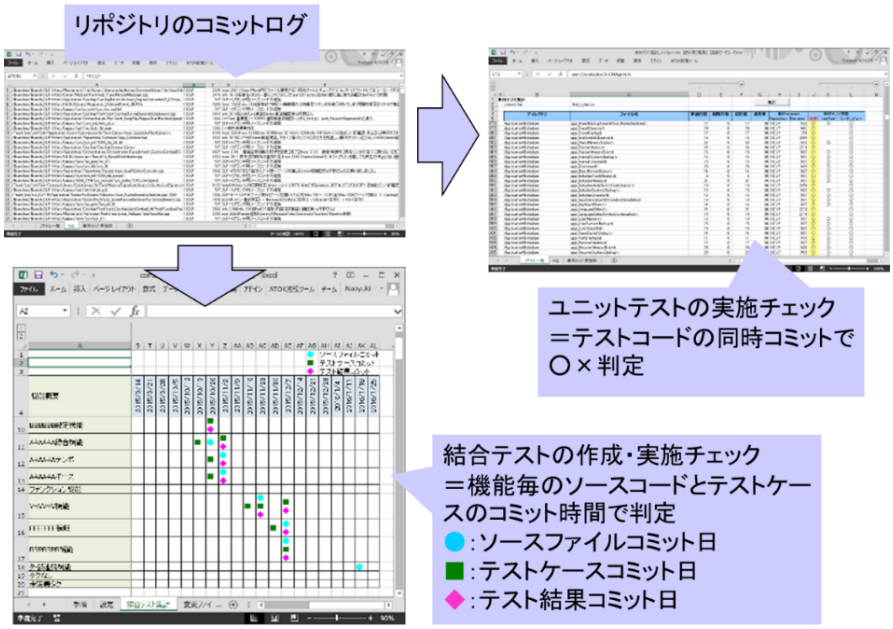


図 72-13　テスト実施状況の自動集計

5. 達成度の評価、取り組みの結果

5.1. プロジェクトメトリクスによる品質特性

本編の取り組みについて、プロジェクトマネジメント観点で当初の目的が達成されているか評価する。同社では 1997 年より 200 以上のプロジェクトで、工程毎の工数や実装行数、バグ密度、レビュー効率、及びそれらの予実績差異など詳細なソフトウェア・メトリクスを収集し続けてきた。また同社の商品は基本となるラインナップが定着しており、同シリーズの製品であれば規模・品質の標準値も存在する。従って本取り組みの適用以前の同シリーズの製品開発のメトリクスと比較すれば、プロジェクトとしての評価が可能である。本取り組み適用の最初の製品出荷を初年度とし、以降 5 年間・全 6 シリーズ 22 機種で適用以前のプロジェクトとの比較を行った。

5.2. 開発工程の改善結果

まず、開発工程の変化について表 72-2 に示す。比較のため、本取り組み開始後に開発した事例適用機種と非適用機種から代表的な 4 機種を選定し、開発工程毎のバグ密度を計測した。非適用機種は、単体・結合テストでの不具合摘出が 25.2%にすぎず、システムテストに負荷が掛かっていたが、適用機種では単体・結合テストに半分以上の工数を割き、47.1%のバグが摘出されるようになった。検証工程の中でも前工程にシフトした結果が見て取れる。

表 72-2　工程毎のバグ密度とバグ件数

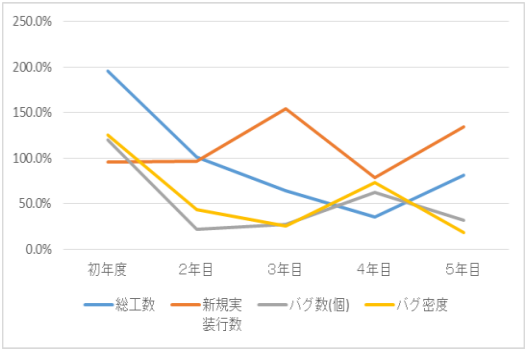
	機種	工数比			摘出バグ件数比		
		結合前	統合	システム	結合前	統合	システム
非適用	機種 W	31.8%	14.1%	54.1%	37.1%	40.4%	22.5%
	機種 X	20.5%	19.5%	60.0%	16.3%	51.0%	32.7%
	機種 Y	24.9%	13.8%	61.3%	11.8%	20.6%	67.6%
	機種 Z	45.6%	15.2%	39.1%	35.8%	20.2%	44.0%
適用	機種 A	51.6%	9.9%	38.5%	62.0%	16.0%	22.0%
	機種 B	61.1%	2.3%	36.5%	67.7%	6.5%	25.8%
	機種 C	54.1%	16.7%	29.2%	36.7%	43.3%	20.0%
	機種 D	46.3%	15.6%	38.2%	21.9%	23.4%	54.7%
非適用機種平均		30.7%	15.6%	53.6%	25.2%	33.1%	41.7%
適用機種平均		53.3%	11.1%	35.6%	47.1%	22.3%	30.6%

5.3. プロジェクト全体での改善結果

次に、プロジェクト全体での品質改善結果を表 72-3 に示す。総工数及び新規実装行数と、それに対するバグ数及びバグ密度変化からも、品質が劇的に向上していることが読み取れる。初年度は CoPaN フレームワークの開発や自動生成など環境構築のための工数が含まれており 195.8%と過去の 2 倍の工数がかかり、バグ密度も 125.2%と以前より増加したが、以後は漸減していることが明らかである。5 年目に本取り組みの最大の目的である商品価値向上のための新機能開発に着手し、ユーザーインターフェースを完全に一新した。新規行数は 134.3%と増加しているが、システムテストにおけるバグ件数は 32.0%に、バグ密度は 18.2%までに激減していることからも、本編の施策がプロジェクト全体に貢献したことが明らかである。

表 72-3　プロジェクト全体の比較結果

年度	総工数	新規実装行数	バグ件数	バグ密度
初年度	195.8%	95.6%	119.7%	125.2%
2年目	101.1%	96.5%	22.2%	44.0%
3年目	64.3%	154.0%	27.3%	25.5%
4年目	35.9%	79.1%	62.5%	73.5%
5年目	81.6%	134.3%	32.0%	18.2%



6. 今後の取り組みと考察

本編では DI コンテナを活用したフレームワーク「CoPaN」を応用したテストフレームワークの展開と、因子の MECE 性に注力した仕様検証法の２点の手法により、検証工数の削減と品質向上を同時に達成したことを解説してきた。

本取り組みの実施は、「2.1.」で述べた通り「銀の弾丸などない」という前提に立ち、全てを変革するという大きなビジョンの下に進められた。約５年にわたる改善活動の間、メンバーのモチベーションが維持できたのは、次の理由によると考えられる。

・ 活動のビジョンを常に示し続けた
・ 活動の全体像を示したことで、個々の改善業務の価値をメンバー全員が理解できた
・ 改善のためのアクションを細分化したことで、小さな改善の積み重ねで常に達成感を得られるようにした
・ メンバーからのボトムアップの提案を受け入れ、改善を加速させた

中でも興味深いことは、この活動自体がボトムアップ的に行われたという点である。「2.1.」図 72-3 にある通り、活動開始時に改善活動の全体像を示してはいるが、本編の内容にはその時点では想定されていなかったものも多く含まれている。ビジョンの共有によってメンバー全員が改善の目的を深く理解したことで、一人一人が自主的に改善活動に取り組んだ。そして、リーダーである筆者がボトムアップの成果を即座に取り入れ、全体の活動に反映することで好循環を生み出す結果となった。また当時のマネージャーは常にリーダーの相談に乗りつつ、長期にわたる活動の支持とリソース確保に努めた。粘り強く成果を待つマネージャーの存在がなければ、本取り組みの実現は不可能だったと思われる。

この活動の前提には、同社で 15 年以上前から詳細なメトリクスを収集し続け、膨大なデータを蓄積してきた経緯がある。これも当時のマネージャーがメトリクスに基づいた管理を継続してきた成果だが、このデータがなければ改善結果を数値化し、改善の過程において周囲の協力を得ることはできなかった。トム・デマルコの「計測できない物は制御できない」[8]という言葉があるが、地道な計測があって初めて改善が可能となる。本編の対象システムは 200 万 step と大規模であるのに対し、改善活動に割り当てられた人員は、開始時は僅か３名、最大でも 10 名以下という非常に限られたものだった。その上、ソフトウェアテスト専門の組織もなく、開発者自らテストを行わなければならない組織体制にある。同社に限らず実際の組込みシステムの開発現場では、少ない人員、期間、そして仕様互換性などの開発に多くの制約条件が課せられることが多い。しかし問題を適切に分析し、一つ一つ課題を解けば実現は不可能ではない。そうした開発現場の改善に本編が一助となれば幸いである。

参考文献

[1] 安立 直之：モデルベース開発における DI コンテナの活用とテスト省力化の取り組み、ソフトウェア品質シンポジウム 2014、2014、
https://www.juse.jp/sqip/symposium/archive/2014/day1/files/happyou_D2.pdf
[2] 安立 直之：「組込システムのモデルベース開発適用における DI コンテナの活用」、先進的な設計・検証技術の適用事例報告書 2015 年度版、2015、
https://www.ipa.go.jp/files/000049369.pdf
[3] Martin Fowler：Inversion of Control Containers and the Dependency Injection pattern, 2004, http://martinfowler.com/articles/injection.html
[4] 鈴木 三紀夫：「Wモデルとは何か」、ソフトウェアテストシンポジウム 2012 東京、2012、
http://www.juse.jp/sqip/library/shousai/?id=52
[5] 飯泉 紀子、田所 孝文、大立 薫、千葉 美千代：「設計者自身による設計品質作り込み ーテストエンジニア視点の活かし方ー」、SQiP 研究会、2009
http://www.juse.or.jp/software/167/attachs/3-2_report.pdf
[6] バーバラ・ミント：考える技術・書く技術―問題解決力を伸ばすピラミッド原則、ダイヤモンド社、1999
[7] Frederick P. Brooks：“No Silver Bullet - Essence and Accident in Software Engineering”, Proceedings of the IFIP Tenth World Computing Conference, 1986
[8] Tom Demarco：“Controlling Software Projects: Management, Measurement, and Estimates”, Prentice Hall, 1986

独立行政法人情報処理推進機構 技術本部 ソフトウェア高信頼化センター（IPA/SEC）