

15-A-12

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¹

1. 概要

本編では、電子楽器システム開発において DI コンテナ技術を適用した、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 DI コンテナ技術適用の経緯及び運用における工夫とその効果についても報告する。

組み込みシステム製品における近年の大規模化・複雑化への対応は、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。その解決策の一つとしてオブジェクト指向技術を基盤とするモデルベース開発の導入事例が増加しているが、理想的な設計モデルを実装に落とすには、モデルとして表現されたオブジェクト間の関連を如何にモデルに影響を与えずに実現するかが課題となっている。また組み込みシステム製品は省リソースのハードウェアで実現する上でパフォーマンスや資源管理など様々な課題が存在する。

そこで同社では 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。現在までに 20 機種以上の同社主力製品に適用し、ソフトウェアプロダクトラインエンジニアリングを実現し、大幅な開発効率・品質向上を成し遂げた。

2. 取り組みの目的

2.1. DI コンテナとは

DI コンテナとは、「DI(Dependency Injection : 依存性の注入)」という考え方に基づき、コンポーネント間の依存関係を実装から排除し、外部の設定ファイルなどを用いて注入するデザインパターンである。Martin Fowler が提唱した概念[1]で以下のようなメリットがある。

- ・ 結合度の低下によるコンポーネント化の促進
- ・ 単体テストの効率化
- ・ 特定のプラットフォーム（OS、ミドルウェア）への依存度低下

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

DI コンテナの基本構造は図 15-A-12-1 の例のように、呼び出し側クラスからは抽象的なインターフェースを呼び出すが、実際に実行されるオブジェクトについては関知しない。予め定義された設定ファイルに基づき、コンテナが依存関係に応じたクラスを選択し実行するという仕組みである。DI コンテナを用いた実装としては、Java の開発環境である Enterprise JavaBeans² などが有名である。しかし組み込みシステムに特化した実装として公開されたものは現時点では開発されていない。

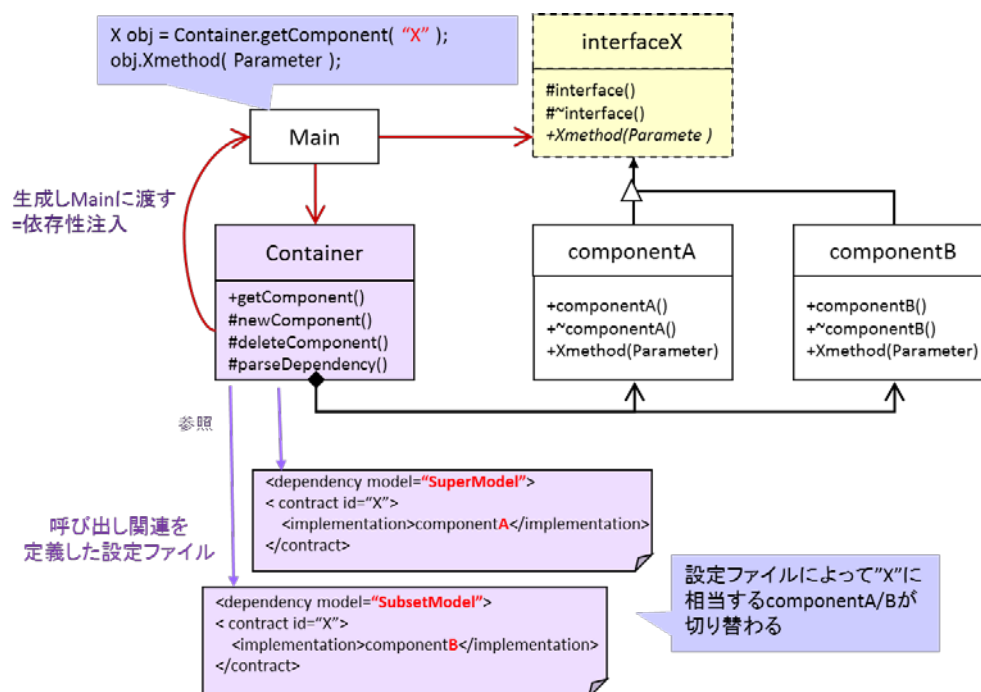


図 15-A-12-1 DI コンテナ基本構造

2.2. 適用前の電子楽器開発の状況

同社はアコースティック楽器で培った音・音楽の技術を背景に、多彩なラインナップの電子楽器を社会に提供してきた。黎明期における電子楽器は、既存のピアノ・オルガンの模倣を目指し、鍵盤を主とした入力操作子と発音原理のみで構成される単純な構造であった。その後多彩な楽器音や自動伴奏といった演奏機能の進化と改良が試みられ、特に 1990 年代以降はネットワーク化への対応や大型ディスプレイの搭載、各種メディアとの連携など、急速な進化を遂げた。一方で楽譜や演奏法、それらを表現する楽曲データ、楽典として電子楽器登場以前から受け継がねばならない音楽理論など、時代によらず不変な仕様も大きい。よって楽器機能を実現するソフトウェアシステムは流用開発を基本とし、大幅な見直しを行う事ができなかった。その結果、2009 年頃にはソフトウェアの複雑化が進行し、再利用性及び開

² Java 言語においてクラスオブジェクトを分散型ネットワーク上で通信を行うためのフレームワーク。
<https://jcp.org/en/jsr/detail?id=318>

発効率が大幅に低下していた。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%と非常に高く高効率な開発を実現していたかに見える。しかし実態は複雑化したソフトウェア資産の調査に多くの工数を割かれたために、新機能開発の工数が減少し実装量が抑制されていただけに過ぎない。また対象プロジェクトには数十機種種の派生開発が含まれていたが、機種間の差分管理はソースコードベースで行われてきたため、機種間の仕様差を一元的に把握する事も不可能な状態であった。その状態を端的に表すのがプログラム生産性である。同社の工数メトリクスから生産性を抽出した結果、図 15-A-12-2 生産性と流用率の通り 2003 年時は約 1,000 行／人月を維持していた生産性が 2009 年には約 500 行／人月にまで低下していたことが判明した。

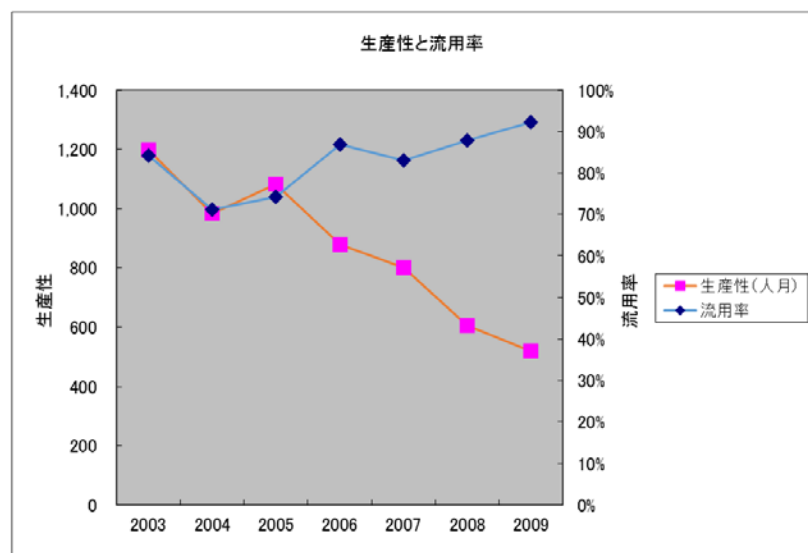


図 15-A-12-2 生産性と流用率

2.3. 課題と目標

前述の通り複雑化したソフトウェア資産のために大幅に低下した生産性を向上し、新機能開発を容易にすることで、商品価値を高め続ける事が最大の目標である。

生産性低下の原因は複雑化したソフトウェア資産そのものにあることが明らかだが、2009 年時点で電子楽器のソフトウェアシステムは約 200 万 step と大規模化しており、同一機能を維持しながらソフトウェアシステムを刷新するには、莫大な工数が必要となっていた。その工数を抑えるためには部分的・段階的なソフトウェアの入替がプロジェクトの制約条件の一つであり、また流用・派生開発の更新頻度を短く、多数のバリエーションを可能にすることが、もう一つの制約条件であった。従って安全に部分入替を可能にする分割統治の仕組みと、流用・派生の変化点（バリエーションポイント）の集約を同時に実現する技術が必須だった。

そこで DI コンテナ技術を活用したソフトウェアフレームワークを導入し、既存のソフトウェア資産を分割し部分入替を可能にすること、またバリエーションポイントと分割されたコンポーネント間の依存関係に集約することを課題とした。

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み込みシステムである。DI コンテナを適用技術の主体とし、ユーザーインターフェースからハードウェア層までシステム全体をモデルベース開発に適合させる。具体的には次の技術を用いる。

- ・ 同社独自開発の組み込みシステム向けフレームワーク「CoPaN」
- ・ UML モデリングツール「Enterprise Architect」
- ・ 反復型開発プロセス

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」以降で解説する。システム改善の評価としてソースコード解析による品質測定と、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

2.1 で述べた DI コンテナのメリットは、モデルベース開発において次の効果が期待されると本論では仮定した。

- ・ モデルベース開発で実現した高品質な実装を、コンポーネント化し再利用性を高められる。
- ・ モデル検証をコンポーネント単位で行うことが容易になる。
- ・ 特定のプラットフォーム(OS、ミドルウェア等)に依存した実装部を分離することで、プラットフォーム依存の記述を排した理想的な設計モデルが可能になる。

しかし 2.1 で述べた通り、組み込みシステムに特化した DI コンテナの実装は公開されていない。そのため同社では DI コンテナを用いたフレームワーク「CoPaN」を独自開発した。以下、フレームワーク「CoPaN」の実装と特徴について述べる。

4.1. DI コンテナを用いたフレームワーク「CoPaN」

4.1.1. DI コンテナの実装

CoPaN は C++言語をベースに開発したもので、Embedded C++の規約に準拠しテンプレートなどの機能を用いずに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 の DI コンテナとしての抽象インターフェースは、以下 3 種類に集約されており、モデル間の関連は 3 種類のみで規定される。これは 4.1.2 で述べる単方向関連を実現するための工夫である。そして 3 種類のインターフェースを束ねる形で 1 コンポーネントに 1 個の Core クラスを定義し、コンポーネントの起動・終了といったライフサイクルを制御している。

- ・ Commander・・・オブジェクトに一方的に伝達するためのインターフェース
- ・ Parameter・・・オブジェクトの情報を参照するためのインターフェース

- ・ Notifier・・・オブジェクトの状態変化を通知するためのインターフェース

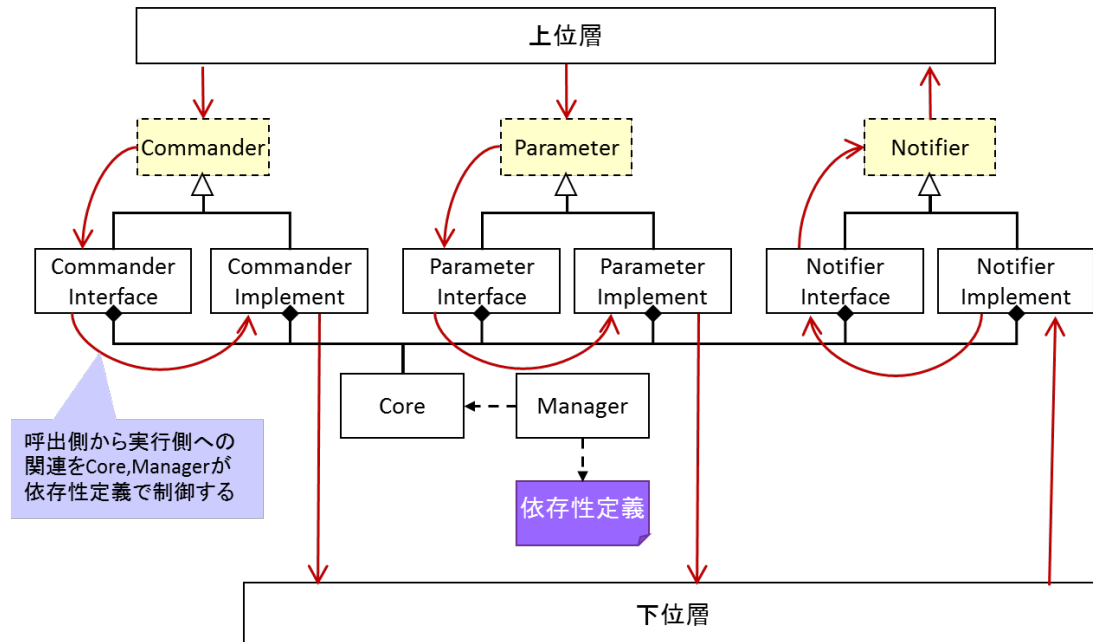


図 15-A-12-3 CoPaN フレームワーク概念クラス図

システム内の全てのモデルは、図 15-A-12-3 の各 Implement クラスを継承することで独自のインターフェースを定義し、モデルから他モデルへの関連はあくまで抽象化された Interface クラスを呼び出すことで実現する。実際にモデルから他のどのモデルが呼び出されるかは CoPaN フレームワークを管理する Manager クラスが、依存性定義に従って決定する。

ここで Commander を用いた実装例を解説する。図 15-A-12-4 にて componentA は“SongPlay”を実行したいとした場合、“SongPlay”を抽象的に指し示す ID を用いて、Interface クラスを取得する。Interface クラスは Commander の抽象メソッド“Send0”のみ定義されており、componentA は“Send0”メソッドを呼び出し指定した曲の再生を促す。CoPaN フレームワーク内部では当該システムの依存性定義の中から“SongPlay”に相当する実行主体を検索し、図 15-A-12-4 のように Implement クラスを継承した componentB を呼び出し、componentB に実装されている楽譜情報を読み出すという流れになる。別のシステムでは、同じ“SongPlay”を呼び出しても依存性定義の違いから componentB+を呼び出し、componentB+に実装されている MP3 データの再生を行う。同様に Parameter、Notifier も依存性定義によって実行主体を置換することが可能である。

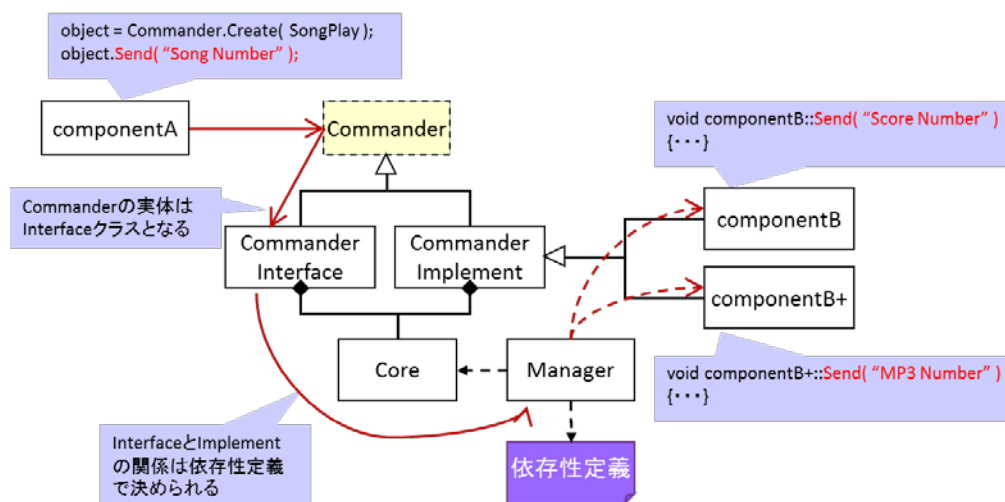


図 15-A-12-4 Commander を用いた実装例

依存性定義は製品システム毎に帳票管理されており、帳票からヘッダファイル (.hpp) の形で自動生成し、システム初期化時にコンポーネントの基本クラスである **Core::Regist()** メソッドでヘッダファイル (.hpp) を読み込み、生成・初期化する **FactoryMethod** パターンを採用している。(図 15-A-12-5) この時点でシステム上のどの ID にどの **Commander Interface** と **Commander Implement** を割り付けるかが決定される。システム初期化後、製品使用中のユーザー検索の結果 **Commander** が呼び出される際には、既に生成済のオブジェクトが再利用される。これによりインスタンス生成と検索に必要な時間を最小化し、低リソースの組み込みシステム上でも瞬時に目的の **Interface** を呼び出すことが可能となっている。詳細は 4.2.2 にて述べる。

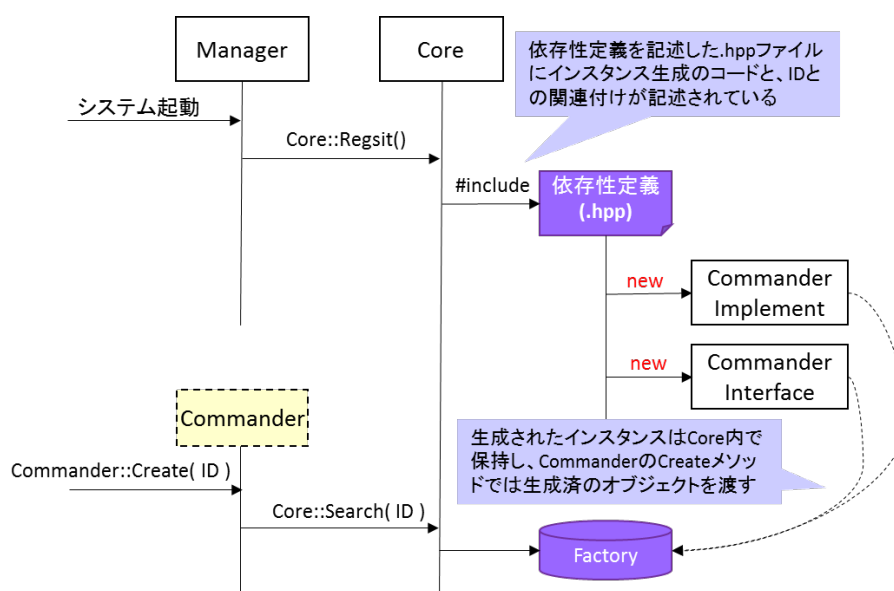


図 15-A-12-5 依存性定義の実装

4.1.2. Observer パターンと単方向関連

CoPaN フレームワークの設計における基本方針として、単方向関連を軸としたシステム構造を目指した。例えば3つのコンポーネント間での関連を例に取り単方向関連に限定したシステムと双方向が可能なシステムとを比較する。単方向関連に限定したシステムでは関連数は最大3だが双方向が可能なシステムでは最大6の関連が生じる。(図 15-A-12-6) 実際、事例適用以前の同社のシステムを調査するとほぼ全てのコンポーネントが双方向関連で結合されており、その上各コンポーネントは異なる複数のタスク上で動作していたため、開発中に度々デッドロックや無限ループなどの不具合が発生していた。

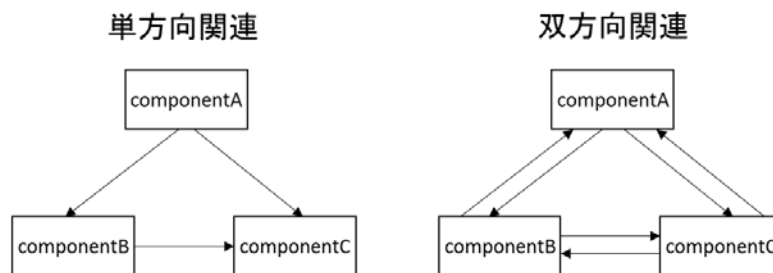


図 15-A-12-6 単方向関連と双方向関連

双方向関連によって生じる問題を削減するため、CoPaN フレームワークは単方向関連のみに限定する仕組みを提供している。ユーザーインターフェース層の componentA がミドルウェア層の componentB を利用するケースを例に取る。図 15-A-12-4 のように componentA は componentB に対し Commander を用いて様々な命令を行うが、componentB の実行結果を知りたい場合がある。その場合、既存のシステムでは componentB から componentA を呼び出し実行結果を伝える事になる。それでは componentB は componentA を知識として持っている事になり、仮にユーザーインターフェースを変更し componentA に修正が生じた場合、componentB にも修正が及ぶ。CoPaN フレームワークではこれを Observer パターンを用いる事で解決した。componentA は予め実行結果を知りたい Notifier に自身の Observer を登録する。componentB は自身の実行により結果が変化したという事のみを Notifier に伝達する。この仕組みによって両者は抽象化されたインターフェースのみを知識として持ち、componentB は componentA に依存することはない。また、Observer での通知は componentA が実行されているタスクコンテキスト上で動作する仕組みになっているため、componentA は componentB のタスクコンテキストを知る必要もない。(4.2.1 参照)

実際の事例適用前後のソフトウェアシステムを Doxygen³ で解析し、コンポーネント間の依存関係を出力した結果を図 15-A-12-7 に示す。適用前はアプリケーション層が 11 個のコンポーネントで形成されており、複雑な双方向関連であったことが分かる。適用後は中央のアプリケーション層は 3 個のコンポーネントに集約された上、完全な単方向関連を示してい

³ ソースファイルのコメントから文書を生成するツール。 <http://www.doxygen.org/>

ることが分かる。なお右端の赤枠は CoPaN フレームワーク本体であり、各コンポーネントが CoPaN フレームワークを介してシステムを構成していることが見て取れる。

ここで一点本編の特徴的な部分について注記しておく。完全単方向関連のソフトウェアシステムが構築可能だったことは電子楽器固有の仕様に依存している。電子楽器は鍵盤や多数の操作子を備えたユーザーインターフェースは非常に複雑だが、基本的には入力に対し何らかの発音原理を伴って音として出力することだけが求められるシステムである。実時間処理は非常に高い精度が要求されるが、制御系装置のように制御対象からの物理的なフィードバックを検知し制御に戻すような要求はないことが特徴である⁴。

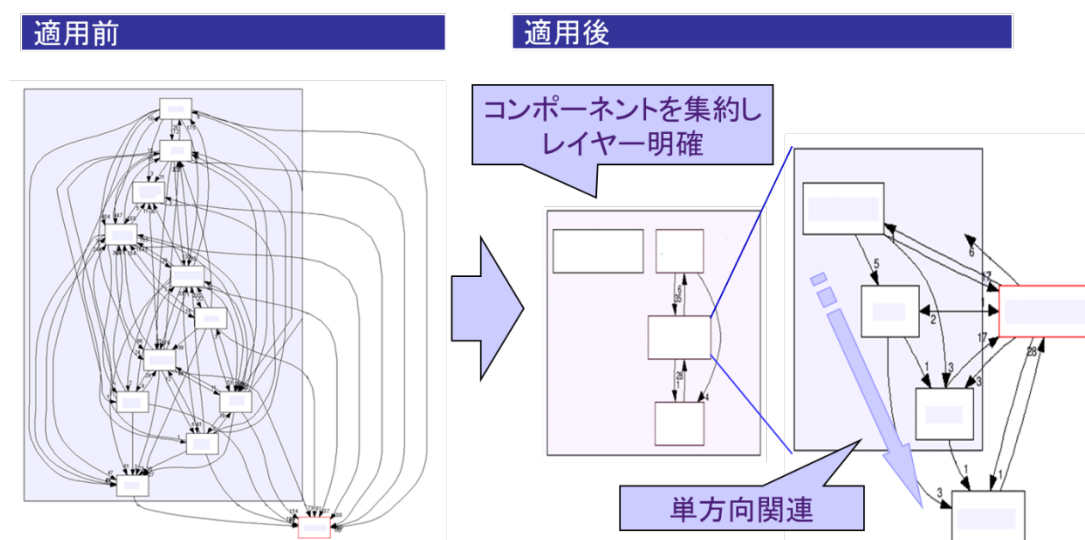


図 15-A-12-7 事例適用前後の階層構造

4.1.3. 汎用データクラスと翻訳機構

4.1.1 で述べた様に CoPaN フレームワークは 3 種類の抽象インターフェースに集約しているが、インターフェース上でのデータのやりとりの方法として汎用データクラスと翻訳機構という概念を取り入れた事が工夫の一つである。

1 点目の汎用データクラスは、電子楽器システム内で扱われるデータを集約し音楽記号クラスとその継承クラスとして実装されている。コンポーネント内ではそれぞれのモデルに特化した情報を有するが、抽象インターフェースへの引数は基本型である音楽記号クラスとして渡され、相手のコンポーネントも音楽記号として受け取る。このため相手のコンポーネントにおける内部型に依存せず抽象インターフェースを利用することが可能となる。

受け取ったコンポーネントが音楽記号から内部型に変換したい場合に必要な仕組みとして 2 点目の翻訳機構がある。(図 15-A-12-8) 音楽記号クラスと対になる形で基本型の翻訳機構クラスとその継承クラス群として提供され、各コンポーネントは継承クラスを用いて内部型

⁴ 同社製品にはスピーカー出力音を再度信号処理し音場効果に加える「iAFC」などの技術も存在するが、総じてフィードバック系は限定的という意味である。

に変換し利用する。また、CoPaN フレームワーク内でタスク間通信やデータアクセスを伴う際にも用いられ、翻訳機構を用いて情報のシリアル化を暗黙裏に行っている。翻訳機構は 4.4.1 の段階的なモデルベース移行において大きな成果を得ている。

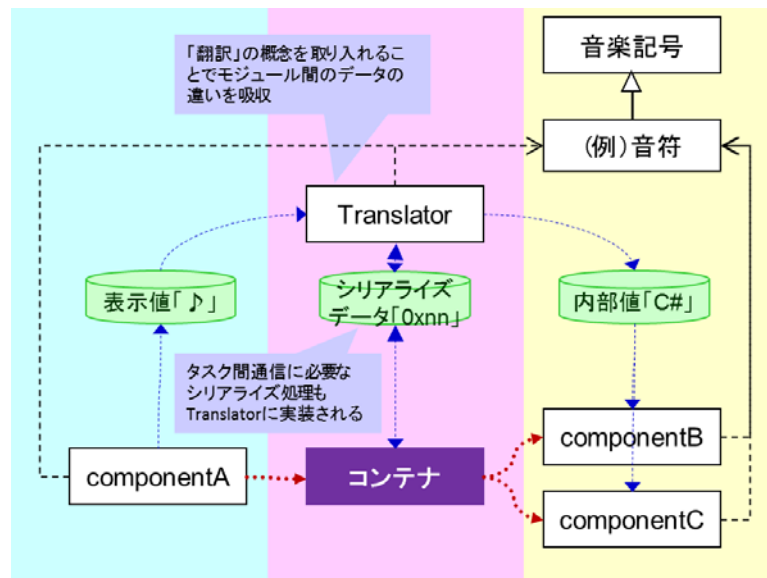


図 15-A-12-8 翻訳機構の仕組み

4.2. 組み込みシステムのための工夫

4.2.1. OS 依存性の内包

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く、関連に必要なコンポーネント間通信も内包している。CoPaN フレームワークの抽象インターフェースは次の 4 種類の通信方法を実装している。

- ・ 直接呼び出し・・・ライブラリコールとして直接相手先のコンポーネントが呼び出される
- ・ セマフォ呼び出し・・・呼び出し時にセマフォロックし、相手先のコンポーネントの処理中はセマフォ管理される。主に **Parameter** の参照に使われる。
- ・ タスク間通信（非同期）・・・メールボックスを利用したタスク間通信。
- ・ タスク間通信（同期）・・・相手先のコンポーネントはそのコンポーネントに割り当てられたタスクコンテキストで実行し、タスク優先度とは無関係に実行完了を待つ通信方法。相手先のコンポーネントの実行中、自身はスリープ状態になる。

CoPaN フレームワークの **Interface** クラスは、通信方法毎に基本クラスを継承して実装されている。全ての関連は依存性定義で帳票管理されているが、通信方法も同じく依存性定義の中で管理されており自動生成される。4.1.1 の図 15-A-12-5 でシステム初期化時に全てのイ

インスタンスを生成するが、その際、依存性定義にはそれぞれどのような通信方法で実現するか記述されており、通信方法毎に内部実装を継承した **Interface** クラスが生成される。4.1.1の図 15-A-12-4 に示した通りコンポーネントは基本クラスの抽象インターフェースを利用するため、その実体がどの通信方法を実装しているかは意識することではなく、設計モデルに通信方法を記述する必要もない。

このように **CoPaN** フレームワークが **OS** 依存の通信方法を内包することで、コンポーネントの関連一つ一つの実行コンテキストの調整を実現している。その結果、組み込みシステム特有のハードウェアに依存したパフォーマンスチューニングや、ハードウェアの変更への対応が容易となった。例えばある製品システムを複数製品に展開する際、その中に低性能なハードウェア資源の製品が含まれていたとする。性能面の課題から **componentB** をタスク間通信からセマフォ処理での実行へと変更する必要が発生したとする。既存のシステムではコンポーネントが直接 **OS** のシステムコールを変更したり、関数の構造自体を見直すことに迫られる場合がある。しかし **CoPaN** フレームワークでは図 15-A-12-9 のように設計モデルや実装に手を加える必要はなく、依存性定義の修正のみで対応が可能である。

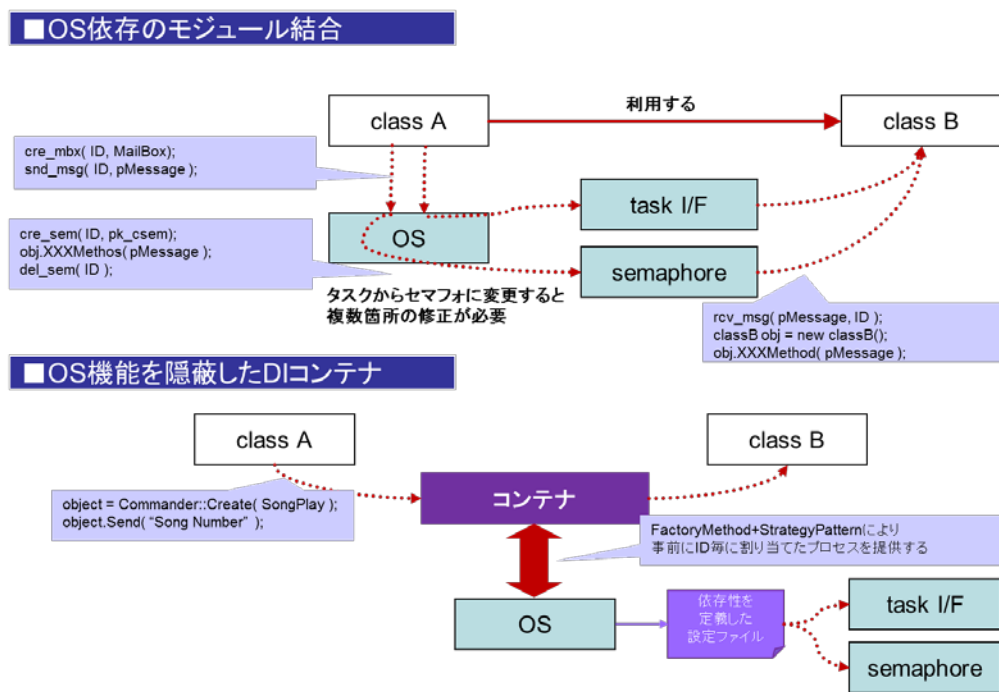


図 15-A-12-9 呼び出し方法の変更例

4.2.2. Factory Method パターンとメモリチューニング

モデルベース開発は UML や数学・物理表現を用いてソフトウェアをモデルとして表現し、ソースコードを限りなく自動生成することが基本である。その際自動生成するソースコードは、組み込みシステムでは C/C++ 言語が用いられるが、再利用性とモデルの忠実性からオブジェクト指向言語である C++ の利用が広まっている。しかし組み込みシステムで C++ を利用する際のデメリットとして、インスタンス生成に伴うメモリ使用量の増大と動的なメモリ確保開放による実行速度の低下がある。インスタンスの生成には大きく以下の 3 種類が存在する。

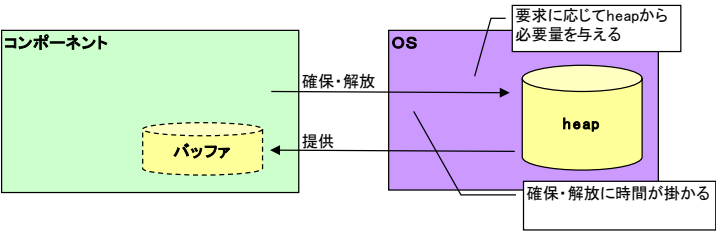
- ・ 大域変数としてオブジェクトを `static` 宣言する
- ・ 局所変数としてローカルスコープ上にオブジェクトを宣言する
- ・ ヒープ領域上に確保するものとしてオブジェクトを `new` で宣言する

このうち、最後の `new` によるインスタンスの生成・開放はパフォーマンスに大きな影響を与える。 μ ITron はヒープ領域として「固定長メモリプール」と「可変長メモリプール」の 2 種類を用意しているが、両者の領域確保/開放時間には顕著な差がある。固定長メモリプールは常に一定時間 (μ sec オーダー) で領域確保/開放が可能であるが、可変長メモリプールは解放時の時間は不定である (条件によっては数百 μ sec オーダーの時間が掛かる)。従ってインスタンスの生成・開放は、いつ・どのような頻度で行われ、どの領域を使用するか気をつけねばならない。

4.1.1 で述べたとおり CoPaN フレームワークは、FactoryMethod パターンを用い、全てのインスタンスをシステム初期化時に生成し、動作中は生成済のインスタンスを利用することで解決を図っている。しかし、この方法はインスタンスの生成・開放をシステム初期化時に集中させているだけであり、製品の起動時間が増大する方向に傾く。またコンポーネントの中には設計モデル上、動的に確保・開放が必要なインスタンスが必要な場合もある。両者の問題を解決する方法として事例においてはシステム起動時及び起動後の定常状態におけるインスタンスのサイズと数を計測し、全てのインスタンスを固定長メモリプールに配置し最適化する方法を採った。

μ ITron におけるインスタンス生成及び開放は、最終的に `malloc()` 関数が呼び出される仕様となっている。本編ではこの `malloc()` 関数にロギング機構を設け、サイズ毎のヒープ使用量を計測した。まずシステム初期化から各コンポーネントの初期化までの使用量を計測し、固定長メモリプールを最適化した。次に電子楽器における基本仕様操作を行い、それぞれの操作におけるヒープ使用量を計測した。例えば演奏を録音・再生する機能があるが、録音した曲のリストアップにおいては仕様上の最大数でのヒープ使用量を計測するなどし、それぞれの操作における最小公倍数となるヒープ使用量にて固定長メモリプールの最適化を実施した。(図 15-A-12-10) この結果、事例適用前の製品相当のシステム起動時間及び実行速度を実現した。

通常のメモリプールの使用方法



最適化されたメモリプールの使用

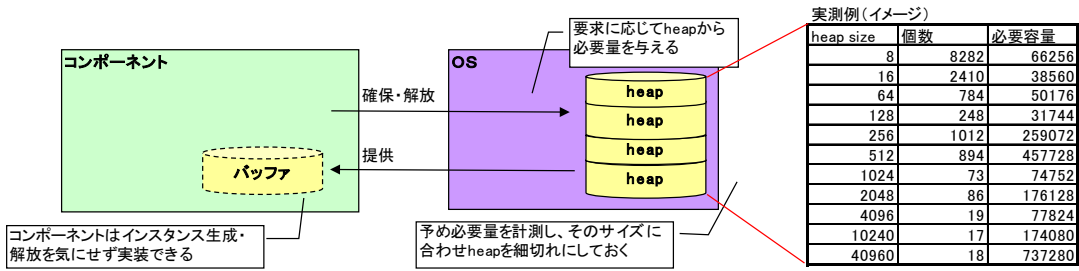


図 15-A-12-10 固定長メモリプール最適化イメージ

4.3. ユーザーインターフェースの自動生成

電子楽器においてユーザーインターフェースは最も重要な価値の一つである。グラフィカルユーザーインターフェースから LED 表示の単純なものまで広く商品展開しており、時には 100 個を超えるボタンやノブスイッチを有する商品が存在する。ユーザーインターフェースの変更容易性は、本事例の取り組みの目的である生産性の向上に大きく寄与する。本取り組み適用前のソフトウェアシステムを調査した結果、ユーザーインターフェースのソースコードが全体の 32%を占めていた。(図 15-A-12-12) 全て開発者が手作業で実装しており、不具合の温床となっていた。ユーザーインターフェースの自動生成は既に多くの製品が存在するが、電子楽器の多様なユーザーインターフェース全てを網羅する例はない。そこで本事例では、CoPaN フレームワークに基づいた表示コンポーネントを基本構造にし、コンポーネントの状態遷移と振る舞い、アプリケーション層との関連の状態遷移表として表し、ソースコードの自動生成を行った。(図 15-A-12-11)

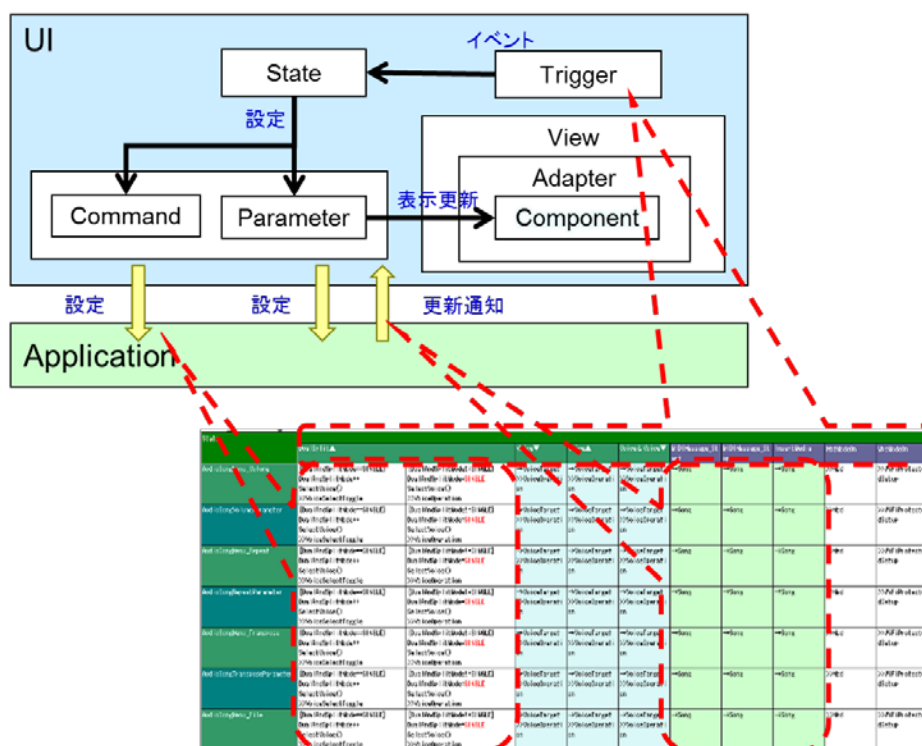


図 15-A-12-11 表示コンポーネントと状態遷移表

状態遷移表では表示器をコンポーネントとして定義し、入出力となる操作子及びアプリケーション層のインターフェースを記述する。操作子及びアプリケーション層は CoPaN インターフェースで抽象化されているため、アプリケーション層の内部実装やデータ型に左右されずにユーザーインターフェースを構築できる点が大きな特徴である。この結果、システム全体に占めるユーザーインターフェースのソースコード比は事例適用後（2013 年時点）で事例適用前に比べて 22%に減少した。また、その全てのコードは自動生成化されている。さらに、状態遷移表で不具合を摘出することが可能なため、ユーザーインターフェースに起因するバグ密度は 0.022 件／kloc と非常に低い数値となった。

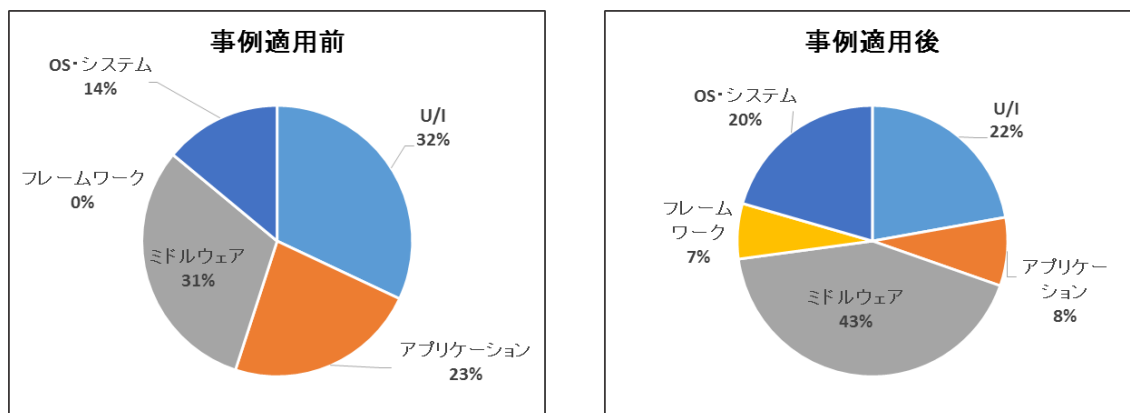


図 15-A-12-12 システム全体に占めるユーザーインターフェースのソースコード比

4.4. アーキテクチャ維持のための施策

4.4.1. 段階的なモデルベース移行

ソフトウェア資産を段階的に置き換える際の課題の一つとして、古いソフトウェア資産のデータ型の扱いがある。モデルベース開発で新しく設計されたモデルにおけるデータ型と、古いソフトウェア資産のデータ型とで相違がある場合、相互に変換が必要となる。その際、新しく設計されたモデルに古いソフトウェア資産の知識を持ち込まれることで依存関係が生じ、理想的なモデルが崩壊する恐れがある。4.1.3 の翻訳機構は 2.3 で触れた段階的なモデルベース開発への移行において効果を発揮した。新しいモデルと古いソフトウェア資産との結合に CoPaN フレームワークを用いる事で、新しく設計されたモデルは音楽記号クラスのみを用いることができ、古いデータ型や実装への依存性を最小限に抑えた。(図 15-A-12-13) 古いソフトウェア資産に対しては翻訳機構を用いて古いデータ型へ変換するため、こちらもソースコードに手を加えることなく再利用が可能である。これにより新しく設計されたモデルを維持しつつ、コンポーネント単位で古いソフトウェア資産を段階的に置き換えていくことが可能となった。

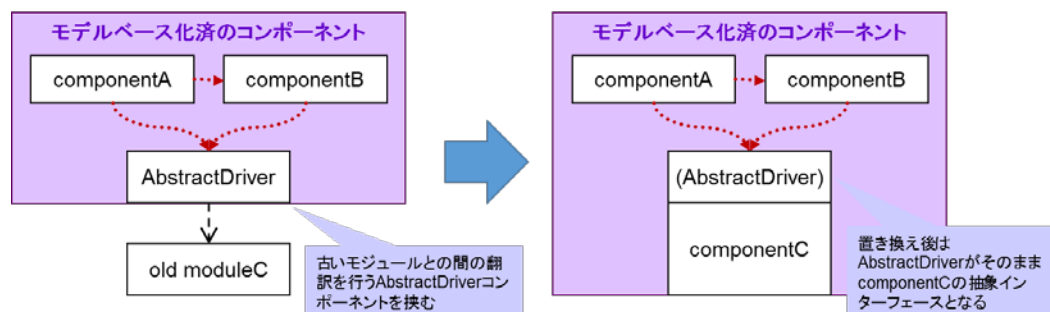


図 15-A-12-13 モデルベースへの置き換えイメージ

4.4.2. UML と実装コードを一致させるリバーズエンジニアリング

本事例におけるモデルベース開発は、開発対象の仕様を UML 記述による抽象モデリングによって、仕様及び設計を明確にし、モデルの段階でシステムの問題点を取り除くというアプローチである。UML 記述でのモデルを作成し、そこからソースコードを部分的に自動生成する手法をとっている。しかし 4.4.1 で述べた通り、開発対象に対し段階的なモデルベースへの移行を行う過程では、新しく設計されたモデルの維持という運用上の課題がある。モデルベース開発の原則はモデルによる仕様の定義を行い、モデルからの自動コード生成により実装を行うことであるが、現実にはプロジェクトや実装上の都合でソースコードを直接修正することがある。その後モデルへのフィードバックを怠るとモデルとソースコードが乖離し、結果としてモデルベース開発が崩壊する。それを防ぐために本事例では UML モデリングツールとして Enterprise Architect⁵ を採用した。Enterprise Architect は UML からのソ

⁵ <http://www.sparxsystems.jp/products/EA/ea.htm>

ソースコード自動生成機能だけでなく、ソースコードからのリバースエンジニアリング機能を有している。同機能を用い、静的構造を表すクラス図のモデルとソースコードの常時一致を実施した。具体的にはプロセスルールとしてソースコードの実装後に必ずリバースエンジニアリングを行い UML の更新を行う事を義務づけた。さらに、設計レビュー及びソースコードレビュー時に、プロジェクトマネージャーが UML 記述のモデルとソースコードのファイルの作成日時を確認することで、実装と完全一致したモデルの維持を実現した。

4.4.3. 反復型開発プロセスによる動作モデルの維持

4.4.2 で述べた通り、本事例におけるモデルベース開発の中心は、UML 記述による抽象モデリングに基づいた開発手法である。最新のモデルベース開発で用いられる MATLAB/Simulink⁶ を用いたシミュレーションを伴うモデルベース開発と異なり、システムの動的な挙動をモデルで検証することが困難である。同じモデルベース開発でもシミュレーションを伴う開発手法と比べ、設計時点で検証することが難しく、実装後に問題が発覚するまでのリードタイムが長くなる傾向にある。そのため本事例では反復型開発プロセスを取り入れることで、モデリングと実装の期間を短くし、手戻りを最小限にした。

反復型開発プロセスとはシステムの基本構造を部分的に開発し、徐々にサブセットを追加することで最終的に全体を構築する開発プロセスである。本事例では通常 1 年間の開発プロジェクトを 6 分割し、2 ヶ月単位で一部分の要求分析～設計～実装～テストまでを行うイテレーションを設定することで、4.4.1 で述べた段階的なモデルベース移行を進めた。(図 15-A-12-14) 要求分析から設計は UML 記述のモデリングで行い、その後実装及びテストにて動作検証を行う。ウォーターフォール型の開発ならば、モデリング後動作検証で問題が発覚するまで数ヶ月後となるが、反復型開発では数週間後には問題が発覚し、次のイテレーションで修正が適用可能となる。この 2 ヶ月単位の基本イテレーションは、開発者個人ベースでは一日単位のスケジュールに分割され、開発者は日々モデリング～実装～テストを繰り返す。非常に短期間のプロセスを開発者全員が平行して進めるため、どこかで不具合が生じると他の開発者の作業にも大きな影響を与える可能性がある。そのため本事例では継続的インテグレーション⁷を導入した。具体的には Jenkins⁸ を用いて自動ビルドし、開発者がコミットしたソースコードに問題がないかシステムの基本動作の確認を行う。1 日 3 回の定期ビルドと内部リリースを行い基本動作を製品レベルで維持することで、開発者の手戻りと問題発生時の影響時間を最小限に抑えることが可能となった。

⁶ <http://jp.mathworks.com/products/simulink/>

⁷ 「継続的インテグレーション入門」ポール・M・デュバル、スティーブ・M・マティアス、アンドリュウ・グローバー

⁸ <https://jenkins-ci.org/>

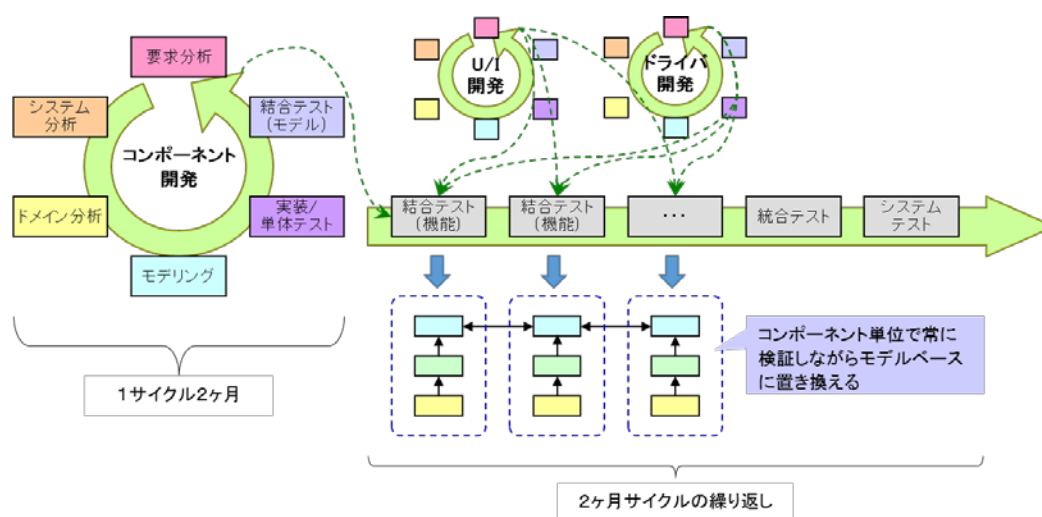


図 15-A-12-14 反復型開発プロセス

5. 達成度の評価、取り組みの結果

5.1. ソースコード解析による品質特性

本事例は株式会社オーグス総研の協力の下⁹、モデルベース開発への移行を推進した。モデルベース開発以前と以後とでソフトウェア上で評価した。

株式会社オーグス総研では、ソースコードの静的解析を行い、ソフトウェア品質特性（信頼性、効率性、保守性、移植性、再利用性）の観点で得点化するツール「Adqua（アドクア）」¹⁰を開発している。同ツールを用い、本事例での取り組み適用前と適用後の解析を行った結果が表 15-A-12-1 である。品質副特性まで含めた結果を考察すると、システムの堅牢性に貢献する信頼性、効率性の指標で大幅に改善していることが明らかとなった。また保守性のなかの試験性も大幅に向上している。この結果からソフトウェアの静的構造からは改善が確認できる。ただし、本事例の最大の目的である再利用性は 57.73 ポイントと非常に低い数値となっている。この点について再検証を行った結果、4.3 で触れたユーザーインターフェースの自動生成が影響していることが判明した。自動生成ツールの都合上、ソースコードとしては難解な部分が含まれていることが原因であり、自動生成対象を除くと 63.48 ポイントと改善が進んだという結果が得られている。

⁹ 08 年度～12 年度事例・ヤマハ株式会社、http://www.ogis-ri.co.jp/casestudy_old/e-01-20.html

¹⁰ <http://www.ogis-ri.co.jp/product/b-08-000001A6.html>

表 15-A-12-1 Adqua による解析結果

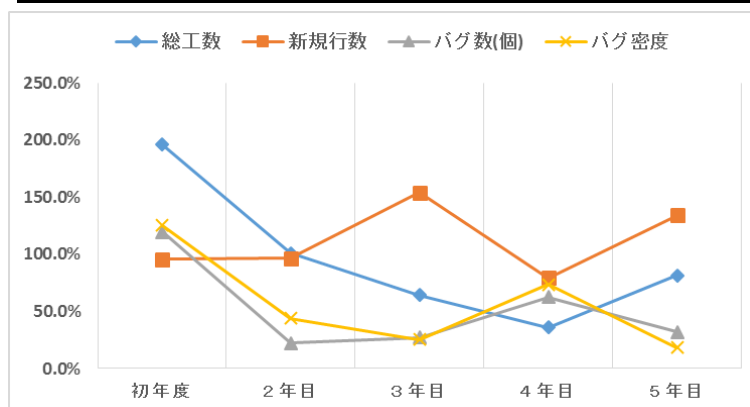
項目		改善前	改善後
信頼性	成熟性	87.07	93.13
	障害許容性	87.43	91.87
効率性	時間効率性	72.27	73.94
	資源効率性	71.13	87.6
保守性	解析性	81.03	83.07
	変更性	75.78	71.83
	安定性	75.6	77.1
	試験性	68.14	72.25
再利用性		62.56	57.43
再利用性(自動生成除く)		62.56	63.48

5.2. プロジェクトメトリクスによる品質特性

次にプロジェクトマネジメント観点で本事例の目的が達成されているかを評価する。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・品質の標準値が求められる。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。本事例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。表 15-A-12-2 の通り、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 倍の工数が掛かったが、以後は漸減していることが明らかである。4 年目に 35.9%まで削減した後、5 年目に本事例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から完全に一新した。それでも総工数は以前と比べ 81.6%に留まっている。また、システムテストにおけるバグ摘出数及びバグ密度は激減している。

表 15-A-12-2 プロジェクトメトリクスと比較結果

年度	総工数	新規行数	バグ数	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%



5.3. ソフトウェアプロダクトライン化による生産性の向上

本編事例は 2010 年に開始し、2014 年時点で 5 年目となる。その間に同社電子楽器製品のうち 6 シリーズ／22 機種に適用された。2010 年以前はシリーズ毎に異なるソフトウェアシステムを採用していたが、全て統一されたシステム上で開発されることにより、大幅なコスト削減効果を生み出した。ソフトウェアプロダクトラインエンジニアリングの基本設計となるバリエーションポイントの考え方は、DI コンテナで管理される依存性定義での変更のみとすることで、ソースコードレベルの流用から設計モデルレベルでの再利用に移行した。5.2 に示した通り総工数は大幅に削減しつつ、新規行数は増加傾向にあり、当初の目的である生産性の向上は果たされている。

6. 今後の取り組みと考察

本取り組みでは組み込みシステム上でモデルベース開発を推進するための基本アーキテクチャとして、DI コンテナを活用したフレームワーク「CoPaN」を開発した。DI コンテナのもつ依存性注入という概念に加え、モデル間の関連の抽象化を積極的に推し進めるため、インターフェースとデータ型の抽象化に取り組んだ。DI コンテナによる関係の外部化に加え、関係性自体を概念レベルに引き上げることで、モデル毎の設計思想を崩さずにシステム全体を構築することに成功している。また、DI コンテナに OS 機能とメモリ管理を取り込むことで、組み込みシステム開発で課題となるパフォーマンスや省資源の課題にも対応している。ただし、この仕組みを維持し続けるには、モデルベース開発に移行していくための綿密なシ

システム分析と移行計画があり、徹底した自動化と継続的インテグレーションが必須となる。

なお、本取り組みは冒頭で述べたとおり、対象システムは 200 万 step と大規模であるのに対し、最大人員 10 名以下という非常に限られたリソースで行われた。さらに、音楽理論を基準に従来製品との仕様互換性を高度に求められる中でのモデルベース開発への移行作業という困難なものであった。事実、初年度の製品は操作性から発音仕様まで従来製品を完全にトレースしつつ、モデルベース開発への置き換えを行った。同社に限らず実際の組み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割し、一つ一つ課題を解けばモデルベース開発への移行は不可能ではない。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、
http://www.juse.jp/sqip/symposium/2014/timetable/files/happyou_D2.pdf
- [2] Martin Fowler : Inversion of Control Containers and the Dependency Injection pattern, 2004、<http://martinfowler.com/articles/injection.html>
- [3] 星暁雄：Java 開発を変える最新の設計思想「Dependency Injection (DI)」とは、日経コンピュータ、2005、<http://itpro.nikkeibp.co.jp/free/ITPro/OPINION/20050216/156274/>
- [4] Paul M. Duvall, Steve Matyas, Andrew Glover : 継続的インテグレーション入門、日経 BP 社、2009
- [5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides : オブジェクト指向における再利用のためのデザインパターン、ソフトバンククリエイティブ、1999

掲載されている会社名・製品名などは、各社の登録商標または商標です。

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)