半導体製造装置のFA間通信ソフトウェア開発
～オブジェクト指向リファクタリングにおける再設計プロセス～

M2007MM007 本間翔大
指導教員：沢田篤史

1　はじめに

我々はOJLとして，半導体製造装置のFA間通信ソフトウェア開発を行った．半導体製造装置のFA間通信ソフトウェアは手続き指向言語で開発され，開発されはじめてから10年以上が経過している．その間の機能追加・変更の繰り返しによる場当たり的なコードの追加・変更によってその記述は散在している．よって，開発者以外の者がそのコードを理解し，修正することは容易ではない．原因は，統一的な設計思想がソフトウェア全体に行き渡っておらず，ソフトウェアの保守性が低下していることが挙げられる．

データ抽象化と多相性の概念を背景にしたオブジェクト指向技術 [1] を適用し，振る舞いを変更せず構造を整理・再構築するリファクタリング [2] を行い，再利用性・保守性を向上する技術がある．我々はこの技術をオブジェクト指向リファクタリングと呼ぶ．しかし，オブジェクト指向リファクタリングには系統的な開発プロセスが定義されていない．

また，ソフトウェア開発において，ソフトウェアアーキテクチャ[3][4] は重要である．ソフトウェアアーキテクチャは初期設計方針を統一し，開発者間の相互理解を深めることができる．また，ソフトウェアアーキテクチャを同様の品質特性と機能要求を持つソフトウェアに転用することで，大規模な再利用を行うことができる．

半導体製造装置のFA間通信ソフトウェアには，同様の品質特性と機能要求を持つ既知のソフトウェアとして，オペレーティングシステム [5]（以下，OS）におけるプロセススケジューラがある．しかし，転用可能なソフトウェアアーキテクチャが存在するにも関わらず，ソフトウェアアーキテクチャに基づいた開発が行われていない．よって，半導体製造装置のFA間通信ソフトウェアは，OSのプロセススケジューラのソフトウェアアーキテクチャに基づいて開発されるべきである．

よって，我々はソフトウェアアーキテクチャに基づくオブジェクト指向リファクタリングの開発プロセスを提案する．ソフトウェアアーキテクチャに基づくオブジェクト指向リファクタリングを行うことで設計思想を統一し，系統的な開発を行うことができる．

本OJLの目的は，提案した開発プロセスに基づいて半導体製造装置のFA間通信ソフトウェアの構造を整理・再構築し，再開発を行うことである．また，提案した開発プロセスにおいて中心となるソフトウェアアーキテクチャを構築することである．

スケジューラのソフトウェアアーキテクチャを転用し，初期設計方針を確立する．そして，仕様との対応関係を明確にすることで対象ソフトウェアのソフトウェアアーキテクチャを構築する．構築したソフトウェアアーキテクチャに基づいて設計を行い，設計に対して少しずつ既存のコードを近づけることでリファクタリングを行う．構築したソフトウェアアーキテクチャを運用し，開発プロセスを繰り返し行うことで対象ソフトウェア全てをリファクタリングすることが可能である．また，オブジェクト指向アーキテクチャスタイルとデザインパターンを組み合わせることで，機能追加・変更に対し柔軟な，保守性の高いソフトウェアアーキテクチャを構築する．

本OJLにおいて習得する問題解決の基礎となるベース技術は，

- ソフトウェア開発プロセス
  - リファクタリング
  - ソフトウェアレビュー
- ソフトウェアアーキテクチャ
  - オブジェクト指向アーキテクチャスタイル
  - デザインパターン

であり，ベース技術を選択，組合わせて適用することで問題を解決する技術であるメタ技術として，ソフトウェア開発プロセスを習得する．

2　提案する開発プロセス

一般的に再開発とは，リエンジニアリングやリファクタリングといった技術を用いて，既存システムと同等，あるいはそれ以上の機能を持つ新たなシステムを作成することを指す．ソフトウェアの振舞いを変更することなく，構造を変更し，保守性を向上する技術としてリファクタリングがある．しかし，既存コードをレビューすることを中心に修正を行うリファクタリングでは，場当たり的な修正により，保守性が向上しない可能性もある．

よって我々は，ソフトウェアアーキテクチャに基づくオブジェクト指向リファクタリングの開発プロセスを提案する．一般的なリファクタリングのプロセスと，本OJLで提案する開発プロセスを，図1に示す．

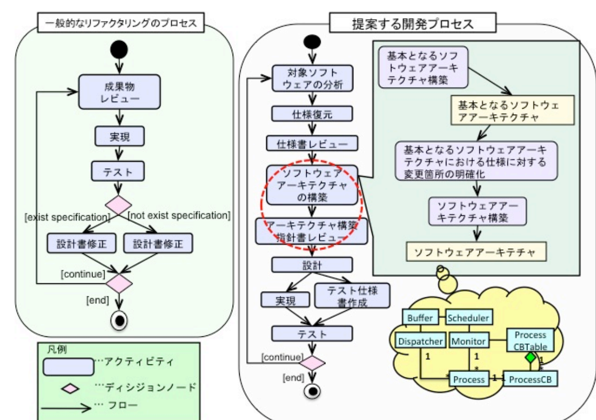

図1　提案する開発プロセス

提案した開発プロセスでは，ソフトウェアアーキテクチャを中心に，既存コードを設計に対し少しずつ近づけていくことでリファクタリングする．構築したソフトウェアアーキテクチャを運用し，機能追加によって開発プロセスを繰り返し行うことで，最終的に対象ソフトウェア全てをリファクタリングすることが可能である．

3　背景技術

3.1　ソフトウェアアーキテクチャ

ソフトウェアアーキテクチャはシステムの構造であり，要素， 可視化された要素の特性，要素間の関係から成り立つ [3]．ソフトウェア開発のステークホルダーとのコミュニケーション，初期の設計方針確立，システムの転用可能な抽象概念において重要な役割をもつ．

また，Documenting Software Architectures:Views and Beyond(V&B)[4] では，ソフトウェアアーキテクチャを捉える視点と各視点の相互作用であると定義している．ソフトウェアアーキテクチャを適切な視点から文書化することで，より明確に記述することを目的としている．V&B ではソフトウェアを捉える視点として 3 つの視点を必要としている．Module Viewtype では静的視点で，Component and Connerctor Viewtype では動的視点で，Allocation Viewtype 物理的視点でシステムを捉える．

本 OJL では，スケジューラのソフトウェアアーキテクチャを転用することで，初期設計方針を確立する．そして，構築したソフトウェアアーキテクチャを各視点で整理・文書化することでプロジェクト内で設計方針を統一すると共に，保守性の向上を狙う．

3.2　デザインパターン

デザインパターン [6] とは，設計の鍵となる側面に名前を付け，抽象化，識別したものであり，プログラミングにおける定石である．品質特性 [7] の向上とそれに対するトレードオフを考慮し，比較・検討したうえでデザインパターンを適用することが重要である．

本 OJL では，オブジェクト指向アーキテクチャスタイルとデザインパターンを組み合わせることで，保守性の高いソフトウェアアーキテクチャを構築する．

3.3　オブジェクト指向によるリファクタリング

データ抽象化と多相性の概念 [1] を背景としてソフトウェアの振舞いを変更せずに構造を変更し，オブジェクト指向コードに既存コードをマッピング [2] する．

本 OJL では，一般的なコードレビュー中心のリファクタリングではなく，ソフトウェアアーキテクチャ中心のリファクタリングを行う．ソフトウェアアーキテクチャに基づくオブジェクト指向リファクタリングを行うことで設計思想を統一し，系統的な開発を行うことができる．

4　半導体製造装置の FA 間通信ソフトウェアにおける専用手法

本 OJL ではプロセスの概念を用いる事で，プログラムの振る舞いの抽象化し，動的挙動に着目した状態遷移機械による振る舞い設計を行う．デザインパターンの State パターンと Command パターンを組み合わせることで状態遷移機械を実現し，階層的な設計を行うことで保守性の向上を狙う．

また，複数のプロセスを並行に動作するスケジューラでは，プロセスの実行管理が重要となる．プロセスの実行状態の管理を実現するために状態遷移機械を用いる．複数のプロセスにおいて同様に実行状態の管理が行われることから，State パターンを適用することで状態遷移機械を実現し，保守性の向上を狙う．

本 OJL では，実開発におけるリファクタリングの制約として，プロセスが実際に行う処理が記述された既存の関数をそのまま利用しなくてはならない．また，長期に渡り使用され，その間に機器の追加や仕様の変更によって追加・変更される関数に対し，柔軟に対応できる構造が望まれる．機能追加・変更に対する保守性の向上を狙い，Factory Method パターンを適用する．

5　ソフトウェアアーキテクチャの構築

本 OJL では提案した開発プロセスに基づき，基本となるソフトウェアアーキテクチャを構築し，仕様に対する変更箇所を明確にすることで対象ソフトウェアのソフトウェアアーキテクチャを構築する．我々が本 OJL において構築した半導体製造装置の FA 間通信ソフトウェアのソフトウェアアーキテクチャを，図 2 に示す．

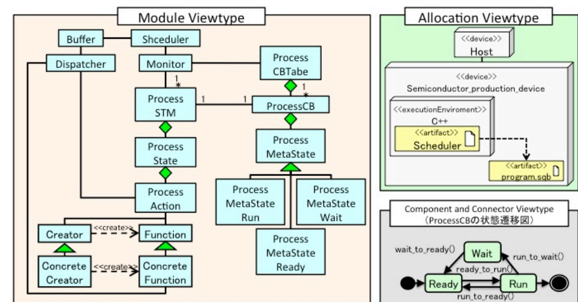

図 2　構築したソフトウェアアーキテクチャ

また，プロセスのブロックとビジーウェイトにより擬似並行実行を実現する際の，プロセスにおける一命令実行時の動的挙動を図 3 に示す．

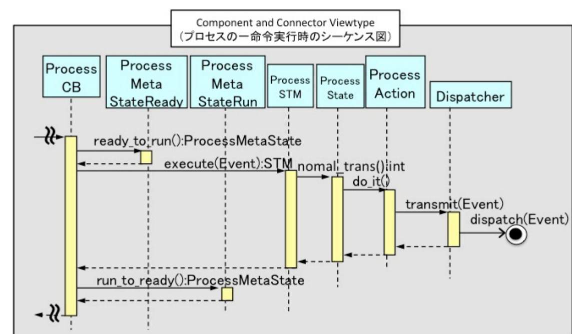

図 3　プロセスにおける一命令実行時の動的挙動

基本となるソフトウェアアーキテクチャの構築と，仕様に対する変更箇所の明確化について述べる．

5.1 基本となるソフトウェアアーキテクチャの構築

提案した開発プロセスに基づき，スケジューラのソフトウェアアーキテクチャを転用することで初期設計方針を確立した．オブジェクト指向を用い構築したスケジューラのアーキテクチャの静的構造を，図 4 に示す．

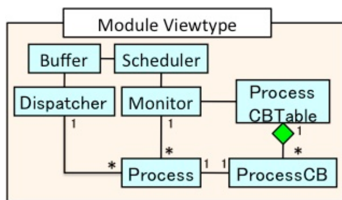

図 4 スケジューラの静的構造

Scheduler がスケジューリングアルゴリズムを実行し，Buffer が受け取ったイベントに対し Process を実行する．Monitor はプロセスの実行を管理し，プロセス制御ブロックである ProcessCB は ProcessCBTable においてプロセス識別子により管理される．実行された Process で発生したイベントを，Dispatcher が適切な場所に通知する．

スケジューラでは，マルチプログラミングによってプロセスを並行に実行する．マルチプログラミングでは，実行中のプロセスから強制的に処理を奪う事で，プロセスの擬似並行実行を実現している．プロセスから処理を奪う方法には，割り込みとプロセスのブロックの二つの方法が存在する．管理するプロセスの特徴に対し選択する必要がある．プロセスのブロックとビジーウエイトによるプロセスの一命令実行による動的挙動を図 5 に示す．

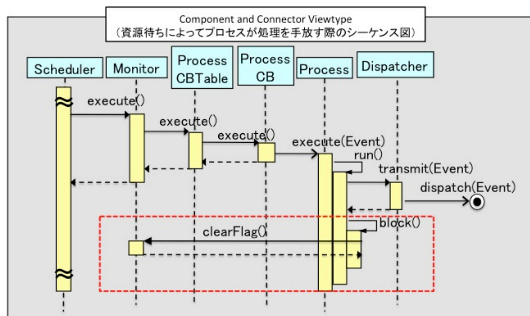

図 5 プロセスのブロック時における動的挙動

5.2 仕様に対する変更箇所の明確化

提案した開発プロセスに従い，基本となるソフトウェアアーキテクチャにおける仕様に対する変更箇所の明確化することで，アーキテクチャを構築する．特に，スケジューラは開発対象毎に管理するプロセスが異なることから，半導体製造装置のプロセスに着目してアーキテクチャに変更を加えた．

半導体製造装置におけるプロセスの特徴は 2 点である．

- 演算装置が処理できる命令レベルでプログラムが記述
- 一命令実行毎に資源待ち

実行が不可分な一命令を実行する度に必ず資源待ちになる装置制御プロセスの特徴から，割り込みを用いる必要が無い．よって，動的挙動としてプロセスは一命令実行の度にブロックし自ら処理を手放す，ビジーウエイトとブロックによる擬似並行実行を選択した．また，スケジューリングアルゴリズムには横取り無し (non-preemptive) の FCFS(First-Come First-Served) を選択した．

我々は，半導体製造装置におけるプログラムの振る舞いに着目し，プログラムのコントロールフローを状態の遷移として表現した．そして，制御を状態遷移時のアクションとすることで，状態遷移機械を用いて振る舞い設計を行った．

状態遷移機械を State パターンと Command パターンを用い階層的に実現することで，各部品の責務が明確になり，状態遷移機械との対応が明確になる．プログラムを基にプロセスを生成する際に，対応関係を明確に表すことができる．状態遷移機械を用いたプログラムの抽象化によるソフトウェアアーキテクチャの変更を，図 6 に示す．

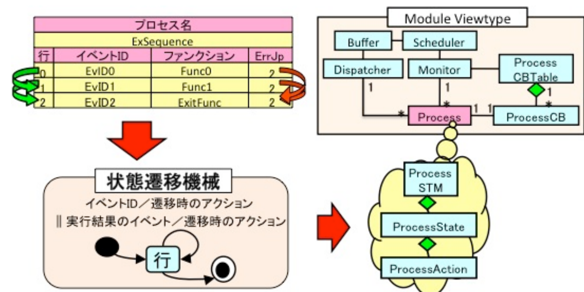

図 6 状態遷移機械によるプログラムの振る舞いの抽象化

また，プロセス制御ブロックでは，全てのプロセスに対し同様に状態の管理が行われることから，State パターンを適用し状態の管理を局所化することで，再利用性・保守性の向上を狙う．既存の関数をそのまま利用しなくてはならないリファクタリングの制約に対しては，関数のラッパークラスの設計変更を行った．ラッパークラスの設計に Factory Method パターンを適用することで，関数の追加・変更に対するコードの変更箇所を局所化し，保守性の向上を狙った．ProcessCB と Process の変更を，図 7 に示す．

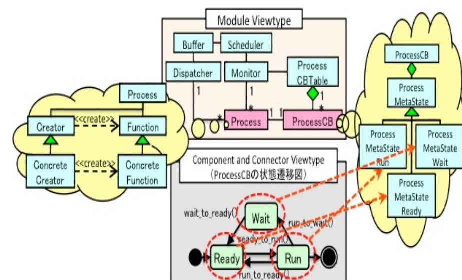

図 7 ProcessCB と Process の変更

6 考察

本 OJL において構築したソフトウェアアーキテクチャの妥当性について，2 点を考察する．

- 適用したデザインパターンと品質特性
- 環境条件の変化への適応可能性

6.1 適用したデザインパターンと品質特性の考察

半導体製造装置の FA 間通信ソフトウェアでは，保守性の高いソフトウェアアーキテクチャが望まれる．本 OJL で

構築したソフトウェアアーキテクチャにおいて，適用するデザインパターンによって向上するソフトウェア品質特性と，トレードオフの観点で比較・検討し，考察する．

本OJLではプロセス設計において，状態遷移機械をStateパターンとCommandパターンを組み合わせることで保守性の向上を狙っている．デザインパターンを適用しない場合とStateパターンのみを適用した場合の比較を，表1に示す．

表1 プロセス設計における比較

デザインパターン	保守性	効率性	信頼性
なし	×	○	○
State	△	△	△
State + Command	○	△	△

StateパターンとCommandパターンの組み合わせにより階層的な設計を行うことで，変更箇所が局所化され，変更箇所が明確になることから，保守性が向上する．しかし，構造が複雑になることで効率性と信頼性が低下する．

また，既存の関数のラッパークラスの設計には，Factory Methodパターンを適用することで，保守性の向上を狙っている．同様に，関数のラッパークラスをプロセスから分離しようと考えた場合，Commandパターンの適用が考えられる．更に，生成規則を分離するためには，Abstruct Factryパターンや，Factory Methodパターンのサブクラス生成にPrototypeパターンを組み合わせる方法も考えられる．表2で各々を適用した際の比較を示す．

表2 既存関数のラッパークラス設計における比較

デザインパターン	保守性	効率性	信頼性
なし	×	○	○
Command	△	△	△
Abstruct Factory	△	△	×
Factory Method	○	△	△
Prototype	○	×	△

Factory Methodパターンの適用は他のパターン適用時に比べ，プロセスから関数の記述とそのラッパークラスの生成規則を局所化できることから，保守性が向上する．更にPrototypeパターンを組み合わせた場合，インスタンス生成後の初期化が必要なことから，効率性は低下する．

保守性の向上を優先し，トレードオフによる効率性・信頼性の低下を最小限に抑えようと考えた場合，適用したデザインパターンは妥当である．よって，保守性の高いソフトウェアアーキテクチャを構築することができたことから，妥当であると考える．

6.2 環境条件の変化への適応可能性の考察

半導体製造装置のFA間通信ソフトウェアにおいて，環境条件に変更があった際の影響について考察する．半導体製造装置のFA間通信ソフトウェアにおける環境条件の変更として，取り扱うプログラムの仕様に変更があった場合，機器の追加による仕様の変更があった場合等が考えられる．

プログラムの仕様に変更があった場合，実行中のプログラムを抽象化したプロセスの変更が必要になる．プロセスを階層的に設計し，プログラムとの対応が明確になっていることから，対応箇所のみの変更を行えば良く，他への影響は少ない．

また，機器の追加が行われる際には，プログラムの追加に加え，プログラムにおける実際の振る舞いが記述された関数を追加しなくてはならない．本OJLでは，関数のラッパークラスにFactory Methodパターンを適用し，プロセスから関数とその生成規則を分離することで，保守性の高い設計を行っている．関数が追加・変更された際に，関数のラッパークラスの追加・変更と，その生成規則の書き換えを行うだけで良い．

保守性の高いソフトウェアアーキテクチャにより，実際に起こりうる環境条件の変化に柔軟に適応な構造を構築できたことから，構築したソフトウェアアーキテクチャは妥当であると考える．

7 まとめと今後の課題

本OJLでは，ソフトウェアアーキテクチャに基づくオブジェクト指向リファクタリングの開発プロセスを提案し，設計に対し少しずつ既存コードを近づけていくことでリファクタリングを行った．スケジューラのソフトウェアアーキテクチャを転用することで初期設計方針を確立し，仕様に対する変更箇所を明確にすることでソフトウェアアーキテクチャを構築した．デザインパターンの適用することで，保守性の高いソフトウェアアーキテクチャを構築することができた．

今後の課題として，構築したソフトウェアアーキテクチャを他のアプリケーションに適用し，実際に再開発することが挙げられる．

参考文献

[1] G.Booch, “Object-oriented development,” *IEEE Trans. Software Eng.*, Vol12, pp. 211-221, 1986.

[2] M. Fowler, *Refactering Improving the Design of Existing Code*, Addison-Wesley, 1999.

[3] L. Bass, P. Clements, and R.Lazman,*Software Architecture in Practice 2nd ed*, Addison-Wesley, 2003.

[4] P. Clements, F. Bachmann, L. Bass , D. Garlan, J. Ivers, R. Little, R. Nord, and J. Staord,*DocumentingSoftware Architectures: Views and Beyond*, Addison-Wesley, 2002.

[5] A. S. Tanenbaum, *Modern Operating System*, Prentice-Hall, 2001.

[6] E. Gamma, J. Vissides, R. Helm, and R. Johnson, *Design Patterns Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[7] 森口繁一, ソフトウェア品質管理ガイドブック, 日本規格協会, 1990.

[8] 野呂昌満, 張漢明, 坂野将秀, 太田将吾, 安江基規, “オブジェクト指向によるSECS通信制御ソフトウェアの開発,” 2007.