Vol.2012-SE-175 No.22
2012/3/16

組込み製品群開発における変動管理へのフィーチャーモデルの適用実験

ーコンパイルスイッチからのフィーチャー抽出ー

牧　隆史[†]　　鈴木正人[†]

複数製品の開発に用いられたソースコード上のコンパイルスイッチを解析することにより製品群の持つフィーチャーを抽出し、フィーチャーモデルをリバースエンジニアリング的に適用することを試みた。さらに、デザインパターンの適用によるリファクタリングの方向性についても検討した。

Experiment of Adapting Feature Model for Variability Management in Embedded Systems

- Extracting Features from Compilation Switches-

Takashi Maki[†]　and Masato Suzuki[††]

By analyzing compilation switches on source codes which has been used for product development over years, we tried to extract hidden features on the products. We also tried to draw feature model for these product. In addition, we discuss feature model oriented refactoring by adopting design patterns.

1. はじめに

類似の複数製品を同一開発プラットフォーム上で開発する場合，多様な機能の実現手段としてコンパイルスイッチ（条件コンパイル指示子）が用いられることがある．コンパイルスイッチによる実現手法はコーディングが容易である半面，実装上の自由度が高いため，十分な検討なしに実装されると後のメンテナンスを困難にする要因ともなり得る．製品毎の機能の相違点(以下変動点と呼ぶ)の設計は本来，求められている要件に従って予め変動要因を特定し，後のメンテナンス性に配慮して変動点の設計を文書化し，その後に実装ステップに進むべきである．（図 1 の実線矢印参照.）

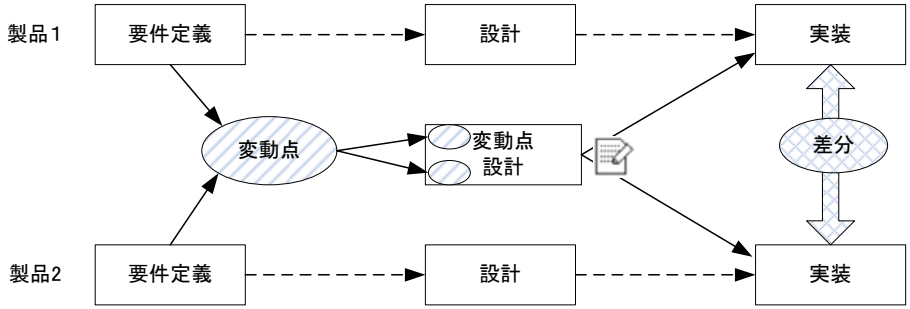

図 1　要件定義・設計・実装の関係

ところが開発プロジェクトによっては，変動点設計の文書を十分に残さずに製品個別に設計および実装が行われてしまう場合もある（図 1 の点線矢印）．開発規模や製品間の変動が小さい場合は実装されたコードそのものがドキュメントの役割を果たせる場合も無いとはいえない．しかしながら，本来あるべき変動点設計の文書が残されていないと，後に設計意図の理解が困難がとなることで，メンテナンスに工数をとられてしまう要因にもなる．とりわけ，変動点がソースコード上のコンパイル SW によって直接に実現されている場合には，情報量が膨大となるソースコードを理解する必要があるため，メンテナンスに必要となる工数は増える傾向がある．本研究では，コンパイルスイッチによって多くの変動点の実装が行われたソースコードを対象として，これらの変動点の間の関係をソースコードから抽出し，本来そのシステムが実現しよ

[†] 北陸先端科学技術大学院大学 情報科学研究科
School of Information Science, Japan Advanced Institute of Science and Technology

うとしていた変動点の関係をフィーチャーモデルの形で文書化することを試みる（図の③の矢印）．さらに，抽出した変動点の実装方法として，デザインパターン[4]を適用してメンテナンス性の高いソースコードに変換する方法についても模索する．

2. コンパイルスイッチの問題点

同一プラットフォームにおける機能の多様性実現手段は，コンパイル時，リンク時，実行時に大別することができる．このうち，コンパイルスイッチよる方法では，コンパイル時に多様性を実現することで実行モジュールが不要な部分を含まなくなるため，実行モジュールのサイズを小さくする目的には有効な手段である．しかしながら，累積的な機能追加の影響によりソースコード上のコンパイルスイッチの数が増えてくると，その弊害も無視できなくなってくる．以下に代表的な問題点について記述する．

・変動点の意図の共有が困難
対象プロジェクトにおいてどの部分のコードが有効なのかを理解するためにはそれぞれのコンパイルスイッチで実現されている設計意図を理解しておく必要がある．コンパイルスイッチの数が増えてくると理解すべき設計意図もそのぶん増えるので，後からプロジェクトに参加したメンバーにとっては理解に時間を要することになる．

・理解容易性への悪影響
コンパイルスイッチは条件コンパイル#ifdef～#else～#endif の指示子であるため，通常の条件文（if～else～endif, swich～case）と同じく実質的な経路複雑度(サイクロマチック複雑度[8])を上げる要因となる．さらに無効化されたコードが残っていることにより保守性が損なわれる．

3. 調査対象システムの概要

本稿では製品開発に用いられたソースコードを調査することで、プロジェクトコード全体のうち相対的にコンパイルスイッチが多用されている構成要素を洗い出し、その出現傾向を分析する。今回分析の対象としたシステムの概要は下記の通りである．

3.1 対象システム

対象のシステムは，民生機器の組み込みソフトウェアであり，機種開発においてはソフトウェアプロダクトライン[1]の考え方に基づき，同時並行的および時系列的な複数製品の開発を同一プラットフォーム上で行っている．ソースコードの規模は開発機種により 50 万行～100 万行程度である．

ソフトウェア構造は図１に示すような３層のレイヤ構造[3]を採用しており，システム全体の制御を行う部分(Ct)およびユーザーからの入力に対応して機器のふるまいを制御する部分(Ui)からなる上層レイヤ，上層レイヤからの指示に従って機器の各機能を実現する中間レイヤ(Func)，そしてハードウェアを直接に制御する部分(Dev)およびシステム全体から共通に利用される部分(Com)からなる下層レイヤに分かれている．レイヤ構造の制約として，上位レイヤから下位レイヤへの参照は許容するが，下位から上位への参照には制約を設けている．また，いわゆる「厳密なレイヤリング」は適用せず，上層レイヤから下層レイヤへの直接参照は存在する．

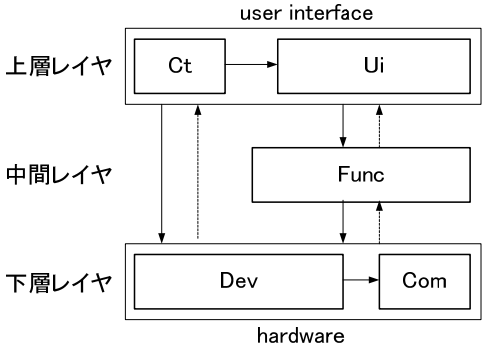

図２．レイヤ構造

3.2 コンパイル SW 内部比率

構成要素ごとのコンパイルスイッチの利用度合いを比較するため、ソースコード中においてコンパイルスイッチで囲まれた部分の占める割合(コンパイル SW 内部比率)を下式に基づいて算出した。

コンパイル SW 内部比率
＝　コンパイル SW で囲まれた部分の行数　／　ソースコード全体行数

図３に、各構成要素におけるコンパイル SW 内部比率の比較を示す。図３の縦軸は各内部比率の平均を１として正規化している。この図から，このシステムでは特に構成要素 Ui においてコンパイル SW が多用されている傾向がうかがえる．これは、機能の差分が主に構成要素 Ui 上で実現されていることが原因の一つとして考えられる．複数機種を同一プラットフォーム上で開発する場合，機種による機能の有無および差異は呼び出し側で選択する方法が直感的にわかりやすいため，依存関係の上位にある構

成要素 Ui においてコンパイル SW が多用される傾向がありそうである．さらに，機能の有無や変更による差異の多様性実現には全体的な視野での設計が必要なこと，また，機能追加や変更の内容は市場ニーズによっても左右されるため予めの予測が難しいことも一因と考えられる．ただし，これはそれぞれのシステムの設計指針によって大きく影響を受ける事項であるため，レイヤ構造を持つ全てのシステムに当てはまるとは言えない．

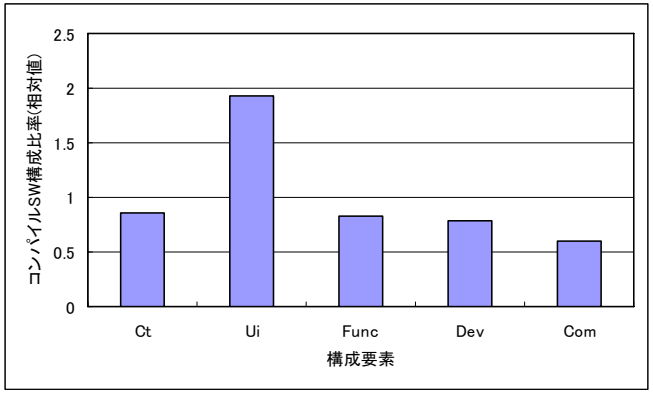

図３　各構成要素のコンパイル SW 構成比率

さらに，構成要素 Ui におけるコンパイル SW 内部比率の時系列的な変化を図４に示す．図４では製品１でのコンパイル SW 内部比率を１として正規化している．当該システムでは機能の異なる複数製品を同時に開発し，さらに新製品の開発ごとにいくつかの機能追加や変更が行われるため，コンパイル SW 内部比率は時間の経過によって増加する傾向がある．図４に示す製品１～製品５は開発時期の順に対応しており，図３に示した各構成要素のコンパイル SW 内部比率は製品５に対応している．利用方法が固定化するなど，変動点としての実装が不要になったコンパイルスイッチについては適宜廃止が行われているのでコンパイル SW 内部比率密度は減少することもある．製品４から製品５にかけての減少はこの間にコンパイル SW の廃止を集中的に行ったことが主な要因である．しかしながら製品１の時点まで戻せているわけではないので，全体としては製品開発に伴って増加の傾向があるものと筆者らは考えている．

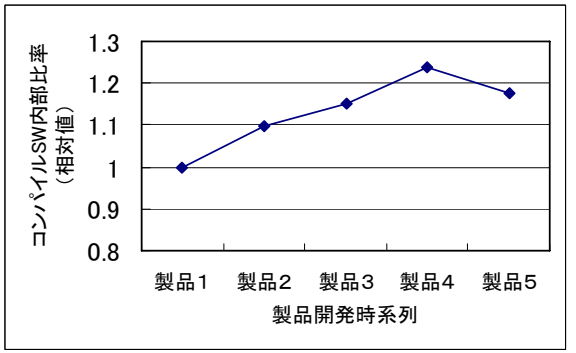

図４　構成要素 Ui におけるコンパイル SW 内部比率の推移

4.　コンパイル SW の利用形態の分類

4.1　コンパイル SW 出現傾向

コンパイル SW が多く用いられている代表的なソースコードのファイルに注目し，コンパイル SW 毎の出現回数を調査した結果を図５のパレート図に示す．

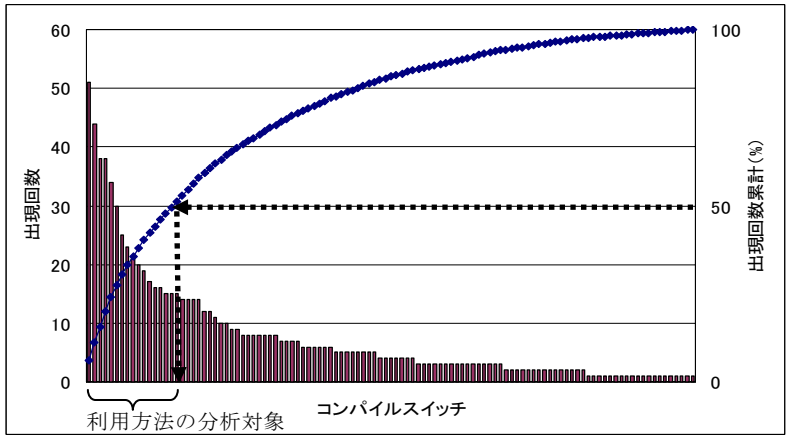

図５　コンパイル SW の出現傾向

分析したファイルは図 1 における構成要素 Ui に属し，その中でも外部への呼び出しを含むものを対象とした．図５の横軸はコンパイル SW の種類を示し、最も出現回数の多いものを左端、出現回数の少ないものを右端として従って順に並べている．棒グラフ(左側縦軸)はそれぞれのコンパイル SW がソースコード上に出現する回数を示し、折れ線グラフ(右側縦軸)は累計の比率を示す．出現回数の少ないコンパイルスイッチも含めた全コンパイルスイッチの出現回数累計は右端で 100%となるが，このソースコードにおいては，出現するコンパイル SW 全体の約半数は出現回数の多い一部のコンパイルスイッチ（図５の左端部分）に集中している．次節では出現回数の累計が 50%までを占める上位のコンパイルスイッチに着目し，コンパイルスイッチの利用方法についてその傾向分析を行うこととした．

4.2 コンパイル SW の分類

図５で洗い出した頻出上位のコンパイル SW についてその利用のされかたを調査したところ，変動点の実現に関係するコンパイル SW の使われ方としては，大まかに下記の３パターンが確認された．

（１） 処理の実行有無を切替えるもの
（２） 処理の内容を切り替えるもの
（３） 分岐を追加するもの

以下にそれぞれについて説明する．

(1) 処理の実行有無を切替えるもの

特定のコンパイル SW がある場合（または無い場合）のみに処理を実行するようにするものである．変動の内容としては処理 A で実現される機能の有無を実現するものに対応する．下の例では COND_1 が定義されているときのみ処理 A を実行しようとするものであり、ここでの処理 A には変数への代入や関数の呼び出しが含まれる．

```
#ifdef COND_1
   処理 A;
#endif
```

(2) 処理の内容を切り替えるもの

コンパイル SW の有無によって呼び出す関数の引数を変えているパターンである．変動の内容としては機能を選択的に切替えるものに対応する．

```
#ifdef COND_2
    処理 A;
#else
    処理 B;
#endif
```

(3) 分岐を追加するもの

コンパイルスイッチにより，特定の条件下での処理が追加されているパターンである．下に示す例ではコンパイルスイッチ COND_3 の追加に伴って，”case STATE_3:” が追加され，その中で変数 variable に VALUE3 が代入されている．ここでは既存 case を含む全ての case 中において同一の変数 variable に値の代入が行われているが，代入する変数が異なる場合や，関数呼び出しを含む場合も存在する．条件分岐が switch 文ではなく if-else によって実現されている場合もこのパターンに分類する．当該システムの構成要素 Ui においては機能の追加をこの方法によって実現している箇所が頻出しているため，次章以下ではこのパターンに着目して分析を行うことにする．

```
  switch (state_var) {
      case STATE_1:
        variable = VALUE1; break;
      case STATE_2:
        variable = VALUE2; break;
#ifdef COND_3
      case STATE_3:
        variable = VALUE3; break;
#endif
  }
```

5. コンパイル SW の関係に基づくフィーチャー抽出

今回分析の対象としたソースコードでは，前述のとおり非常に多くのコンパイル SW が用いられているため，変動点管理を効率的に行うためにはまずこれらの関係を系統立てて整理することが重要となる．ここでは製品群開発の観点からフィーチャーの抽出を行うため,まずは出現回数の少ないコンパイル SW に着目してコンパイル SW 間の関係を確認した．その手順を以下に示す．

[フィーチャー抽出の手順]

1. 出現回数の少ないコンパイル SW ひとつに着目する．
2. 1.で着目したコンパイル SW が用いられている関数ひとつに着目し，そこで使われている他のコンパイル SW をリストアップする．
3. 2.で着目した関数内において，1.で着目したコンパイル SW で囲まれた部分で変更されている変数(または呼び出されている関数)と、2.でリストアップした他のコンパイル SW で囲まれた部分で変更されている変数(または呼び出されている関数)が同一であるか調べる．
4. 1.で着目したコンパイル SW が出現する 2.の関数以外のすべての箇所について，出現するそれぞれの関数内において，3.と同様に 2.でリストアップした他のコンパイル SW で囲まれた部分で変更されている変数(または呼び出されている関数)が同一であるか調べる．
5. 4.の結果がすべて同一であれば，2.でリストアップしたコンパイル SW は同一の関心事に関連付けられるとみなし，共通のフィーチャーを割り当てる．

上記手順を形式的に記述するため，ソースコードのモデルを以下のように定義する．

$U = \{ F_i \mid F_1,F_2,\ldots,F_n \}$ // U: ソースコード全体
$F = \{ B_i \mid B_1,B_2,\ldots,B_n \}$ // F: 関数, B: #ifdef ブロック
$label(B_k) = L_k$ // L: コンパイル SW のラベル
$all\_labels(F_k) = \cup_{Bk \in F} labels(B_k)$
$feature\_of(L) = \{ P_i \mid P_1,P_2,\ldots,P_n \}$ // P: フィーチャー
$assign(B_k) = \{v \mid v$ は B_k で代入される変数（左辺に出現）$\}$
$call(B_k) = \{f \mid f$ は B_k から呼ばれる関数$\}$
$assign(L_k) = U_{Bm}\ assign(B_m)$ ただし $B_m \in F \cap label(B_m)=L_k$
$call(L_k) = U_{Bm}\ call(B_m)$ ただし $B_m \in F \cap label(B_m)=L_k$

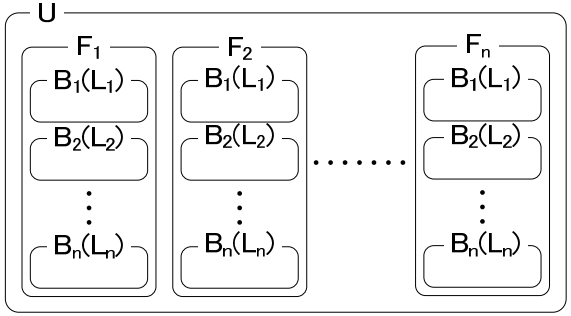

図６　ソースファイル全体と#ifdef ブロックの関係

フィーチャー抽出の手順 calc_feature()はこれらの定義を用いてプログラム 1 に示す擬似コードで表すことができる．

```
calc_feature(Li) {   /* ステップ 1 */
    feature_of(Li) = {Po}
    forall(Fk∈U) {
        if(Li∈all_labels(Fk)) {   /* ステップ 2 */
            target_labels(Fk) = all_labels(Fk)-{Li} /* ステップ 3 */
            forall (Lj∈target_labels(Fk)) {   /* ステップ 4 */
                if( call(Li) == call(Lj) ∧ assign(Li) == assign(Lj) ) {
                    feature_of(Lj)+= Po   /* ステップ 5 */
                }
            }
        }
    }
}
```

プログラム　1

(1) 単一関心事の実現に対応する場合

下記に示す例は複数個のコンパイル SW が上記「処理条件と処理の追加」の形で同一 switch 文中に出現するケースである．異なるコンパイル SW によって囲まれた case 文の中で同一変数への代入が行われている場合，これらのコンパイル SW による変動は同一関心事に関連付けされていると考えられる．これをフィーチャーモデルで表現すると図６のようになる．図６において L93,L95,L110,L24 はそれぞれ同一 switch 文中に出現するコンパイル SW であり，括弧()内の数字はそのコンパイル SW が当該ソースコード上で出現する回数を示す．L93,L95,L110,L24 ともそれぞれ実質的に一箇所のみであるため，一つの Parent Node に関連付けされると考えてよい．このプロジェクトにおいて L93,L95,L110,L24 はいずれも独立して定義可能であるため，これらは独立した optional ノードであると見做せる．なお，この例では switch 文全体が別のコンパイル SW(L86)で囲まれているため，Parent Node は関心事として L86 に対応しており，optional ノードとなっている追加機能はいずれも L86 と集約の関係にある．

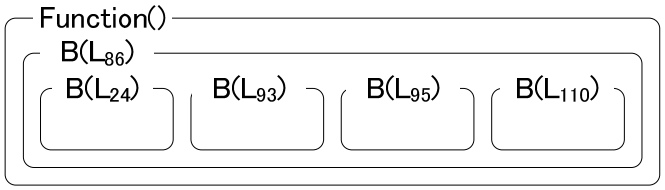

図７ コンパイル SW が特定の一関数内にしか存在しない場合

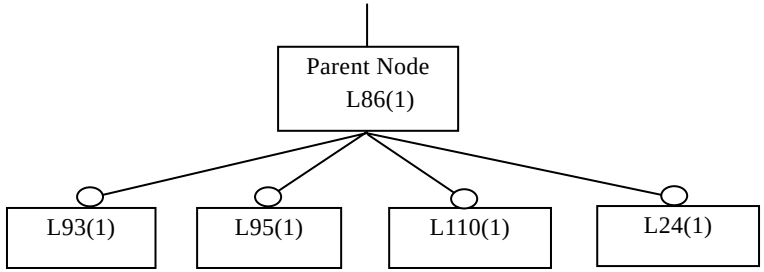

図８ 親ノードとの関連

(2) 複数関心事の実現に対応する場合

上記(1)では各 optional ノードがとなるコンパイル SW がいずれも一箇所だけの例であったが，一般には同一コンパイル SW は複数箇所に出現する．図にその例を示す．ここでは L29 と L34 がいずれも 8 回ずつ出現し，L34 は L29 の中に入れ子になっているので，SW34 による変動は SW29 の子ノードと考えてよい．また，L29 が出現する 8 箇所の switch 文には L2，L6 が必ず並存し，L29 と L2 では同じ変数に代入が行われているが，L6 については L29 では代入していない変数へも代入が行われている．したがって，L2 と L29 は同一親ノード(Parent Node 1)に関連付けされるものと考えてよいが，L6 については別とみなすべきである．なお，L2 は全部で 44 箇所あるので，L29 と同一親ノードに対応した 8 箇所以外の箇所については別のコンパイル SW と関連を持つ可能性がある．

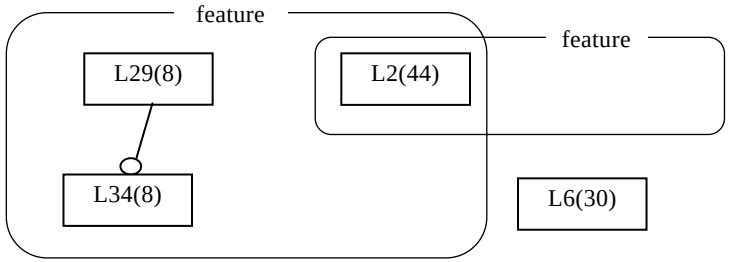

図９ 複数フィーチャーとの関連

以上，4.2(3)に示した swich-case 文に着目してコンパイル SW 間の関係に着目した結果，case 文の部分に適用されている 36 種類のコンパイル SW のうち 17 種類のコンパイル SW に対応して合計 5 個のフィーチャーが抽出された．このことは，個別に管理されている 17 個のコンパイル SW は 5 個のフィーチャーに関連づけてそれらの関係が整理できることを意味している．図１０に，抽出されたフィーチャーとコンパイル SW の関係を示す．

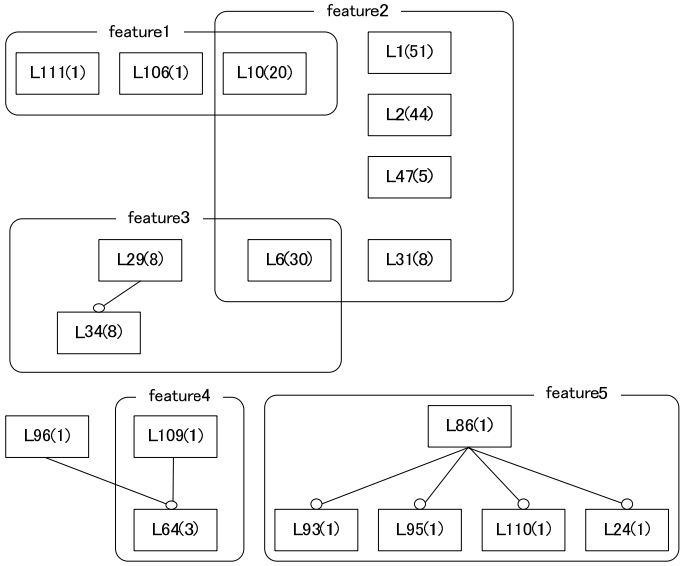

図１０ フィーチャーとコンパイルＳＷの関係（抜粋）

6. フィーチャーの実現

前章まで抽出したフィーチャーに基づいて，メンテナンスのしやすい形にソースコードを再構成することを試みる．

6.1 フィーチャーに基づくソースコードの再構成

コンパイル SW の使われ方は主として 4.2 章に述べたような 3 通りのパターンが見いだされているが，今回はフィーチャーに基づく再構成に着目し，4.2 節の分類における(3)のパターンを対象に再構成を試みた. (3)のパターンのうち再構成の効果が期待できるケースとして，同一のコンパイル SW が複数箇所で同時に同じ使われ方で出現するケースがある．このようなケースにおいては，複数箇所に入れられている似たコンパイル SW を，フィーチャーという括りで再構成することにより，理解容易性や保守性に向上が期待できる．今回の修正方針としては，ソースコード上で使用されるコンパイル SW の総数を減らすことを目標の一つに置き，従来はコンパイル SW で実現されていた部分を基本的には変数で制御する方法に変更する方針で検討を行った．

6.2 デザインパターンの適用の可能性

5 章(2)に示したような，機能追加に伴って条件分岐がコンパイル SW と共に追加されるケースにおいては, 類似の問題を解決するデザインパターン[4][5]を適用することで，見通しのよいソースコードに変換できる可能性がある．具体的には，デザインパターン[4][5]において，複数箇所に類似した条件分岐が存在し，機能追加が必要になった場合に全ての条件分岐に条件を追加しなければならなくなる手間を設計的に回避する方法が STATE パターンとして提唱されている. たとえば既存の実装がプログラム 2 に示すように分岐の追加（4.3(3)のケース）が行われている場合について検討する．下記では，新機能に対応するため新たなケース文が追加され、その分岐部分がコンパイルスイッチ DEF_1 で囲まれている。

```
void main_process() {
   /* 略 */
   int x = getState();
   switch (x) {
      case state_0: func_0(); break;     /* 状態 0 の時の処理内容 */
#ifdef DEF_1
      case state_1: func_1(); break;     /* 状態 1 の時の処理内容 */
#endif
     ...
   }
     ...
 }
```

プログラム 2

プログラム 2 の事例に STATE パターンを適用して構造を変えた例を以下に示す。この構造を用いることで，状態の追加が行われた場合でも main_process()側はその変更の影響を受けないようにすることができる．プログラム 3 では，プログラム 2 事例で DEF_1 で囲まれた case 文は class State_1 で定義されるクラスに対応している．class State_1 のコードは main_process()および他のクラス定義とは独立しているので，class State_1 の追加は他に影響を与えることなく実装することができる．運用方法によっては class State_1 の部分を別ファイルにすることも可能である．

```
void main_process() {
   State *s = new State_0() // 初期状態
   /* 略 */
   s->func(); // 状態に応じた処理
}

class State {
      int x; // 状態変数
           public:
virtual void func() = 0;
}

class Stete_0 : public State {
      public:
      virtual void func () {
            /* 状態 0 の時の処理内容 */
       }
}
class State_1: public State {
      public:
       virtual void func() {
            /* 状態 1 の時の処理内容 */
       }
}
```

プログラム 3

デザインパターンは本来オブジェクト指向処理系での適用を前提としているが，組み込み製品の分野ではまだ通常の C 言語など非オブジェクト指向処理系が使用されている場合も少なくない. 今回の分析対象プロジェクトも C 言語のシステムであるため，実際には上記の C++疑似コードと観測同値となる操作を C 言語において実装する．C 言語によるオブジェクト指向の実装については，たとえば COOL[6]が知られているが，オブジェクト指向処理系の動作を忠実に再現しようとしているために実装負荷が重く

なる傾向があるため，実装時には機構を簡略化するなどの工夫が必要である．

ここではデザインパターン適用可能性について STATE パターンを取り上げて検討したが、対象によっては変動点を呼び出し側から隠蔽する機能を持つ STRATEGY パターンや COMMAND パターンも適用できる可能性がある。

6.3 フィーチャーモデルとの関係

今回の検討では，コンパイル SW の解析からフィーチャーの抽出およびリファクタリングの検討を行った．ソースコード上の個別のコンパイル SW は細分化された機能差分実装の実現手段であるのに対し，複数のコンパイル SW を横断的に見ることで，共通のフィーチャー間の関係、フィーチャー構造が明らかになる．たとえば入れ子になっているものについては，内側にあるものは外側の条件に集約される関係があるため，外側の条件の下でのオプショナルと見なすことができるなどである．本手法は，これをシステム全体に適用することで，最終的にはシステム全体のフィーチャーモデルを構築する糸口になり得るものと筆者らは考えている．

7. 結論と今後

本稿では，ソースコードに含まれるコンパイル SW を解析することによるフィーチャーモデルの適用可能性について検討を行った．その結果，出現傾向が特定のパターンを示すコンパイル SW についてはソースコードの情報を元に，フィーチャーモデルの形式で変動間の関係を表現できることが確認された．これにより，個別に実装されている複数の変動を同一関心事へ関連づけ，ひとまとまりとして管理することが可能となった．

さらに，デザインパターンを活用したリファクタリングの可能性についても模索した．C 言語でのリファクタリングについては，Garrido[7]らも指摘のとおり，条件コンパイル指示子の存在により一般に困難とされているが，今回はオブジェクト指向におけるデザインパターンの観点からアプローチすることを試みた．具体的には，機能追加に伴って Switch-case 文にコンパイル SW と共に case エントリを追加するようなケースについては、switch-case 文自体を STATE パターンの適用によって再設計することで，追加部分をそれぞれ独立したソースファイルとすることができ，結果としてこの箇所に関してはコンパイル SW を使わずに済ませることが可能となった．このようにして解決可能な箇所から順にコンパイル SW を少しでも減らすことで，全体としてのコード全体の見通しがよくなり，次のリファクタリング機会の検討が期待できる．

なお，今回の分析ではコンパイル SW がソースファイル毎に局在していることを前提として，プロジェクト全体の中では代表的なソースコード１つについてファイルスコープでの分析を行った．しかし一般に同一コンパイル SW は複数ファイルに存在し得るので，厳密にはプロジェクトスコープでの分析が必要である．また今回の分析は手作業で行ったが，大規模プロジェクトにおいて現実的な時間内にフィーチャーモデルを起こすためには，ツールによる分析の自動化は必須と言える．これらについては今後の課題としたい．

参考文献

1) P. Clements and L. Northrop: Software Product Lines: Practices and Patterns: Addison-Wesley (2001)
2) R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and S. J. Carriere: The Architecture Tradeoff Analysis Method, Software Engineering Institute, Technical Report CMU/SEI-98-TR-008 (1998)
3) L. Bass, P. Clements and R. Kazman: Software Architecture in Practice(2nd Edition): Addison-Wesley (2001)
4) E. Gamma, R. Helm, R. Johnson, and J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley (1995)
5) J. Kerievsky: Refactoring to Patterns, Addison-Wesley (2004)
6) www.sage-p.com/process/cool.htm
7) Garrido, A. and Johnson, R., ' Evolution in source code: Challenges of refactoring C programs', in *Proceedings of the international workshop on Principles of software evolution,* (2002)
8) N. E. Fenton and S. L. Pfleeger: Software Metrics: A Rigorous and Practical Approach, PWS Publishing Company (1996)