Generative Programming — Methods, Tools, and Applications

2003.9.3 by 久野

1 Forward by James Coplien

□ Coplien さんというのは…パターンとかの有名な人

□ この本はどういう本? (歴史上の位置付け)

□ この 10 年で色々なものが出た

- Subject-Oriented とかコンポーネントとか色々
- どれも「非オブジェクト指向的」側面
- ようやく共通の土台らしきものが見えてきた

□ 共通の土台とは…

- metaobject protocol, reflection, intentionality, insightful interpretation of compoinents, crosscutting
- 要するに→古典的なモジュールの分解

□ GCSE'99 --- 1st international symp. on Generative and Component-Based Software Engineering → 多くのアイデアの結び付きが見えるように

□ 現在が歴史の代わり目かも…(プログラミングと設計手法の分野で)

□ 産業界は O-O パラダイムの制約から逃れる方向を模索

- パターンは一例だけど、Alexander のようになれるとは思えない

□ O-O が出た時に熱狂のあまり色々捨ててしまったかも→今見直しの時期?

□ 革命的変化は難しいけど漸進的なものなら…

□ O-O の「モノ」から concepts, features へ

- intentional programming, domain engineering
- vulgar (non-Alexanderian) pattern movement, AOP, generic programming, multiparadign design
- OOPSLA でも基本機能よりコンセプトがテーマに
- OOPSLA は没落して GCSE がポピュラーに (そうなの?)
- この動きを軽視しない方がいいですぜ (そうなの?)

□ そういうわけで、本書はこの新しい動きに関わるものをまとめている

□ さまざまな手法類を統合的に捉える (それぞれの差異を述べるのでなく)

- components, objects, aspects, reflection, intentional programming, generic programming, domain engineering

□ 一見ごった煮みたいだが「things beyond objects」ですぜ

□ 集めただけでなく統合しようともしている

- ドメイン工学とメタプログラミングが中心となる
- それを中心に集めるだけでなく、統合によるシナジー効果も

□ 新しいモノ (reflection, metaprogramming, aspects) はオブジェクトに取って代わるプログラミングの中心概念になるかも

- たとえそうならなくても、これらが有用であることは確実

□ さあ、あなたもご一緒に (???)

2 From Handcrafting to Autometed Assembly Lines

□ ソフトウェア工学への期待…

- 複雑なものを、高い生産性、高い品質で提供し、保守や変更もしやすく
- …全然ダメ。産業革命以前の「手工業」みたいなもの?

□ Generative Programming では手工業から現在の産業みたいに…必要なこと:

- (1) 単一のシステムを作る→複数のファミリを作る ← 正しいコンポーネントの把握
- (2) 実装の組み立てに generator を使って自動化

3 From Handcrafting...(2)

□ 例: 自動車を自分で部品から組み立てることを考える

- 部品のいくつかはぴったり会わなくて加工が必要かも
- 産業革命の時にもそういう問題を克服するのに何十年か掛かった
- たとえば STL からぴったしな部品が入手できたとしても、組み立てるのは手作業

□ 代わりに「ベンツ S クラス、オプション全部つきで」と言うだけにしたい ← Generative Programming の目標。必要なことは:

- (1) 実装コンポーネントが生産品のアーキテクチャにフィットするように設計されていること
- (2) 構成知識をモデル化し、抽象的な要求を特定部品群の組み合わせ方に変換できること
- (3) 構成知識を generator として実装できること

□ 自動車産業では実際そういうことができている

4 NOTE: 製造ラインに関する誤解

□ 多くの人は製造ラインは「全く同じものを大量に作る」と思っている

- それは実は嘘で。自動車は何千種類もの組み合わせから選ばれるので同じものは 2 台とないと言ってよい
- 実際は「多数の個人の注文」→「それぞれの注文品の製造」→「各ディーラーで引渡し」
- 膨大な生産/流通システムによってこれが可能となっている→ソフトの場合と良く似ているでしょう?

5 産業革命

□ 手工業からどのように変化してきたかを見てみる (図 1-1)

- 互換性のある部品→生産ライン→自動化生産ライン

5.1 互換性のある部品

□ 1785 に Thomas Jefferson がマスカット銃を「互換性のある部品を組み立てて作る」ことを思い付く

□ 1798 に Eli Whitney が銃の大量生産 (4000 丁) を請け負う

- (1) 水力駆動の工作機械、(2) 互換性のある部品の大量生産、(3) 熟練していない普通の人が働く
- 結局、1800 年までという契約に 9 年遅れ、互換性部品ではなかった

□ しかしこの様子を知った同業者がその方向で努力

□ John Hall が 1826 年に互換性のある部品から成るライフルを生産

□ Simeon North が 1834 に Hall のライフルの互換部品を製造

□ 1845 に複数の兵器製造業者が 1841 モデルのライフル部品を製造

5.2 自動車産業における製造ライン導入

□ 自動車→ 1880 年代ころ現れる。1885 年にダイムラーの自動車

- 製造工程は時間が掛かり少ししか作れないものだった
- Ransom Olds が 1901 年に製造ラインの考えを導入 (ライフルと同じ)
- Henly Ford が 1908 年にフォード T 量産開始→自動車の普及
- 1913 年にはベルトコンベア式のラインを導入→生産量が 3 倍以上に

5.3 自動化された製造ライン

□ ロボット: GM 社が 1961 年に導入 (Unimation)。柔軟性がなく非効率。1975 年くらいまでコストに見合わなかった

□ 1980 年代→マイクロチップの「頭脳」を持つ柔軟性のあるロボット→産業ロボットブーム (別の要因:日本の台頭)

- 初期のロボット:大量ロットの加工、現在のロボット:小量〜中量でも使えるように (ジョブショップ方式との競合はまだある)

6 Generative Programming(GP)

□ Generative Programming とは…

- ソフトウェア工学のパラダイムであり、
- ソフトウェアシステム「ファミリ」のモデル化を基盤とし、
- 基本的な/再利用可能な実装部品+構成知識によって、
- 要求仕様から特化/最適化された中間/最終製品を自動生成するというもの。

□ 単一のシステムではなくシステムファミリに焦点

- 各システム (メンバ) は「1 から書く」代わりにファミリに共通の genrative domain model(5 章) から生成

□ システムファミリのモデルは 3 つの構成要素から成る

- (1) メンバの仕様記述手段
- (2) 実装部品 ← これが実際のコードになる
- (3) 構成知識 ← メンバの仕様と生成されるメンバの間のマッピング

□ 自動車の購入でも同様

- (1) 自動車の注文システム
- (2) 自動車を構成する個々の部品
- (3) 注文からどのように車を組み立てるかの知識

□ 仕様記述=problem space、実装部品+可能な構成=solution space(図 1.2)

□ 構成知識 (configuration knowledge) が定めるもの

- システム機能の許されない組み合わせ (例: カブリオレ×サンルーフ)
- デフォルトの設定 (例: サンルーフを指定しないなら普通の屋根)
- デフォルトの依存関係 (例: DC モータの電気自動車→ミッションなし)
- 最適化
- 組み合わせの知識 --- どの部品とどの部品を組み合わせるとどのような機能群を提供できるか

□ 自動車→構成知識の多くは自動組み立てラインの形となっている

□ GP では→構成知識をプログラムとして固定したい

□ 従来のソフトウェア工学→最終製品は作るけど「どうやってそこに到達したか」という生産時の知識は捨てている

- プログラマや設計者は何か考えてやっているがそのことは残されない
- そのため保守も改良も難しく高くつく

□ GP ではこれらの知識をできるだけ捕獲しておきたい

- 生産時の知識 = 構成知識 + 測定ツール + テスト戦略/計画 + エラー診断 + デバッグ機能 + プログラム視覚化 + その他もろもろ
- 具体的に何と何か→ドメイン固有。再利用可能なライブラリとしてパッケージする (active library)

□ active library がパッケージしているものとは…

- 汎用の/ドメイン固有の再利用可能な抽象化を、プログラマが実際にそれらを使うときのコードと一緒にパッケージする
- つまりプログラミング環境をドメイン固有な形で拡張
- 具体例→ Intentional Programming(IP) システム (11 章)

□ IP では「特定のプログラミング言語」の代わりに「言語を抽象化する active library」を使う

- Java とか C++とか言語を決めてしまうとその言語が決めた抽象化を使うしかない
- active libarry では問題 (ドメイン) に合った抽象化を使える
- マルチパラダイムでドメイン特化したプログラミングサポート

□ フレームワーク、コンポーネント ← ソフトウェア再利用において最も効果的と (現在) 考えられている技術

- しかし→どの OOA/OOD もフレームワークやコンポーネント「の開発」は取り扱っていない
- GP では→開発を 2 つのサイクルに分割
- (1) generative domain model の設計と実装 ← 再利用のための開発
- (2) generative model を用いた具体的なシステム生成 ← 再利用に基づく開発
- この (1) も (2) も「1 つのシステムを作るだけのプロセス」(例: 統一プロセス) とは全然違う
- 再利用のための開発のスコープ → システムファミリ

- 再利用に基づく開発 → (1) の投資を活用すべく注意深くデザイン

□ 「システムファミリを対象とする開発」が重要な特徴

□ (1) ファミリ (ないしドメイン) のスコープを決める

- どの機能は入りどの機能は入らないか
- 関与者は誰でそれぞれの目標は何か

□ (2) ファミリ内で共通の機能/変化する機能を同定

- 分析の結果→ feature model(4 章) … UML なんかよりいい
- FM は変化点とそれらの間の依存関係を明示的に記述する→これに基づいてファミリに属する個別システムを生成
- FM はメンバ内の変化とメンバ相互の変化を区別→「太った」コンポーネントやフレームワークを避ける (通常の OO ではアプリ内の変化にもアプリ間の変化にも同じポリモルフィズム機構を使っている)
- FM は実装から独立した「変化点の表現手段」→どうやって変化させるかは後で決められる --- 分析時には扱わないでよい (UML では「ここは集約」「ここは継承」とかいきなり書いてしまうのでダメダメ)

□ (3)generative domain model の設計

- メンバの記述方法
- 共通アーキテクチャ(実装用コンポーネントなども含む)
- 構成知識 --- メンバの記述と組み立てられた実装との対応関係

□ (4) モデルを実装する

- コンポーネント技術や (再び)GP 技法を使う

□ 実装コンポーネント←コードの重複がなく、できる限り柔軟に組み合わせ可能

- STL などの generic programming が持つ特徴 (6 章、7 章)
- GP ではさらにコンポーネントの組み合わせ部分も生成する (generator)

□ 例:下三角行列を高率よくしかも境界チェックつきで扱いたい

- generator は複数の実装コンポーネントから適切なものを選ぶ
- また組み立ても generator が行う
- ということは、generic programming は GP の 1/3 の部分に相当

7 NOTE: コンポーネント

□ コンポーネントの考え方はよく知られている…

- どのコンポーネントは何に使うかも (例:煉瓦は家を作る時に…)
- コンテナなら STL、GUI なら Gtk+、分散なら CORBA…

□ これを一般化→コンポーネントの古典的定義

- たとえばオブジェクトの古典的定義→これこれの機能と属性を持つ (identity, state, behavior)
- コンポーネントの古典的 (人工的) 定義は「有害」

□ コンポーネントは「自然概念」(natural concept)

- natural concept には古典的定義は合わない
- たとえば「机」をどう定義します?
- 例: 「天板に 4 つの脚がついているもの」
- 4 とは限らない。あまつさえ脚がないものが日本にはある (え???)

□ じゃあテーブルとは天板のこと?

- 天板とは水平で平らな作業用平面?
- 平面とは限らないし作業に使うとも限らない…→古典的定義はナイ
- でも多くのテーブルは役に立つ
- 脚がないのに慣れてないならそれはあなた向けでないだけのこと

□ コンポーネントも同じこと

- 必要なコンポーネントのすべての性質や機能を列挙するなんて無理
- ファミリのために必要なコンポーネントが用意できたりその場で作り出せたりすればいいわけじゃない? (たぶん)
- そういう風な文化的変化が今後必要とされている

8 Generative Programming (Cont'd)

□ 今日のプログラミング手法→機能単位 (functional component) をみつけてオブジェクトなりモジュールなりとして実現する

- しかしある種の側面 (エラー処理、同期、永続性、安全性) →局所的に実装できない、今日の OO と合わない
- AOP(8 章) →このような横断的側面をアスペクトとして分離
- アスペクトは weaver によって OO の骨組みに組み込む→実装ができあがる
- OO では細かいメソッドの断片になってしまうものを AOP だとうまく扱える

□ チェック、最適化、weaving、コンポーネントの組み立て→メタプログラミング (=プログラム自身を操作するプログラミング) が必要

- generator(9 章) もメタプログラム ← プログラムを組み立てるから
- generator は階層的かも (下請けの generator を呼んだりするかも)

□ これまでのコンポーネント→「必要なものを探す」ことに焦点

- 名前程度しか手がかりがないのであまりうまく行かない

□ これに対し、generator では「必要なものを作り出す」ところが違う

- 利点 1: オンデマンドで生成するから「山のように用意」が不要、その元になるもの (小さい) があればいい
- 利点 2: 「必要なもの」を指定するので抽象度があげられる、具体的なアルゴリズムやデータ構造は (たとえば技術の進歩により) 変わっていってもいい ← クラス名名指しでは不可能なこと

□ generator はどうやって作るのか?

- (1) 一から作ってもよい (が、大変)
- (2) 言語のメタプログラミング機能を活用 (例:C++ のテンプレート)
- (3)IP などの extendible programming environment を使う ←この場合、言語だけでなくツールや環境まで変化させられる

□ extendible programming environment では、新しい抽象化を作るごとにプログラミング言語がその分だけ拡張されたと考えることができる

- 従来の言語→手続きやクラスなどを定義することで拡張可能
- 多くの言語では構文や最適化などは変えられない
- extendible programming environment ではツールやエラーメッセージなどもドメインに特化して変えられる

□ 自動プログラミング (Automatic Programming) … 古くからあるアイデアだが、GP と似ている面もある

- AP では「一番上の抽象化」からコードに落すことだけ考える →ドメイン知識などが膨大になってしまい難しい
- GP では自動化のレベルは様々に選べる → 実用的

□ GP は「正しい」コンポーネントを作る手段でもある

- 「コンポーネントを手で組み立てることができるなら、そのプロセスを自動化することもできる」(automation principle
- 自動化は魔法ではなく適切なアーキテクチャができれば着実に進められる
- どれくらい自動化できるかはどれくらいドメイン知識獲得にコストを掛けるかによる
- ロボットは大量生産用から少量生産でも引き合うようになった (← CPU 制御のおかげ)
- 同様に、GP の自動化も技術の進歩に従って敷居が下がるはず

9 Benefits and Applicability

□ GP の基本アイデア: システムファミリを対象に generative model を構築し、モデルから具体的なシステムを自動生成

- 実装コンポーネントや構成知識→自動生成ごとに再利用
- 新しい実装が必要となったときにす速くできるかも
- economy of scope

- 実績のあるコンポーネントの再利用→品質、信頼性

□ コンポーネント/ライブラリ/フレームワーク製造が仕事だったら?

- generator は作らなくてもファミリを想定したアプローチは有効

□ コンポーネント開発→ファミリ内の共通点に加え変化点も同定

- domain scoping、feature modeling →これらの共通点/変化点発見に有効

□ GP は構成知識の獲得を伴う→より高い抽象度での再利用やサポート

- generator や active library →ドメインに特化したサポート

□ テンプレートプログラミング→ generator などの実現に有効

□ メタプログラミング環境→ GP により充実したサポートを提供可能

- 例: DMS(9.7.1 節) や IP(11 章) など
- 自動リファクタリング
- 通常の言語での記述より多くの設計知識を固定可能

□ より重要な利点として、投資を保護してくれること

- ある言語で投資したものは言語を移行するとパーになることが多い
- DMS や IP では高レベルの知識をそのまま別の言語に持って行ける (ふーん)
- 「半自動リターゲット」

□ テンプレートプログラミング→ generator などの実現に有効

□ とはいえ、ファミリを作るために用意するのは単一システムよりはコスト高

- だから実際にいくつかのメンバを作る予定がないと引き合わない
- どれくらいから引き合うかは場合によりけりだが見積りはするべき
- 急いで 1 つだけ作るなら GP ではない方がいいかも
- そのドメインを継続的にメシの種にするなら GP にした方がいい
- 1 つ作るだけでもコンポーネントが入手できたりすれば有利

□ GP も他の新技術と同様、導入するなら iterative かつ controlled に

- useful だが failure-critical でない適切な大きさのドメインに適用するパイロットプロジェクトからはじめる
- 一番上位レベルの自動化を目指さなくてよい← GP は「どれくらいの自動化」というレベルが選べる
- generator を開発しなくても、feature modeling や構成知識のドキュメント化は有用
- テンプレート、メタプログラミングは generator のために使っても特定の最適化された実装のために使ってもよい
- DMS や IP が使えれば大きな効果もあるかも知れないが開発プロセスにも大きな変化が及ぶ

□ GP はプログラミングのさまざまなレベルにおいて適用可能 --- scoping, feature analysis, generation → コンポーネントレベル、システム全体レベル以外に手続きやクラスの生成にも使える

□ GP は言語やパラダイムとは直行している --- マルチパラダイムデザインの考え方を受け継いでいる