

ソフトウェアパターンの分類とその記述言語の設計

奥村 和恵[†] 金澤 典子[†] 塚本 享治[†]

[†]東京工科大学大学院 バイオ・情報メディア研究科 メディアサイエンス専攻

簡単に変更でき保守性が高いシステムを作るために、ソフトウェアパターンが提唱された。しかし慣れない設計者がパターンを利用するにはその数が多く、どれを使えばよいか判断が難しい。一方慣れた設計者はパターンを設計に適用する手間が煩わしい。誰もが設計過程でパターンを簡単に選択し、設計図に適用できるツールの開発を目指している。そこで本稿は約 80 のパターンについて目的と実装時の構造を、「目的」「クラス構造」「振舞」の 3 種に着目して分類した。分類結果に基づきパターンを記述する言語構文を定義し、記述実験を行った。

Classifying software patterns and proposing a pattern definition language

Kazue Okumura[†], Noriko Kanazawa[†], Michiharu Tsukamoto[†]

[†] Tokyo University of Technology, Graduate School of Bionics, Computer and Media Science, Media Science Program

The software patterns were proposed for designing systems. But there are too many patterns to select. An expert designer annoys by applying patterns to diagrams. So we are developing an tool for everyone can use patterns easy.

First, we analyzed about 80 patterns, and classified these from three view points –pattern's objective, classes structure, and messages structure. Second, we designed and experimented a pattern defining language from classifying results.

1. はじめに

システムの設計を補助する手段にソフトウェアパターンがある。よく使われる設計の定石を集めたもので、効果的な設計ができ品質を一定に保つ効果がある。

しかしパターンはレイヤーやアーキテクチャに応じて多く種類があり、どれを用いればよいか選択が難しい。またクラスや振舞等の形は決まっており、設計者は図にパターンを適用する手間が煩わしい。

そこで本稿は最適なパターンの選択を手伝い、設計図への記入補助を行う設計支援ツールの開発を目指している。

アプローチとしてまず POSA や GoF デザインパターン、J2EE パターン、EJB パターンなどから約 80 パターンを収集し分析した。そして目的などで分類し、分類結果から専用言語を定義してパターンを定義した。

パターンの情報は UML 図やテキスト形式が多い。設計支援ツールでパターンを利用する

ために、専用の記述言語を定義しデータ化を図った。

2. ソフトウェアパターンの分析と分類

パターンはレイヤーやアーキテクチャに対応して、数多くの人が提案している。[1]は横軸をアーキテクチャパターンやデザインパターン、イディオムというレイヤーで分類し、縦軸を「作業の組織化」などの体系により分けている。

[2]は[1]よりも詳細に、具体的な目的によって適切なパターンを選択できるよう案内している。例えば「データベースと相互作用を行う」項目でも、どのオブジェクトを用いるにより案内されるパターンが違う。

本稿では[1]と[3]～[7]の文献から、代表的と思われる約 80 パターンを収集した。

- POSA パターン…[1]
- GoF デザインパターン…[3]
- J2EE パターン…[4][5]

- EJB パターン…[6]
- その他…[7]

1 パターンにつきシーケンス図とクラス図の 2 図を集めた。しかし特徴的な一方の図だけが記載されている場合が多かったので、集まった図は以下となった。

- クラス図=約 55 図
- シーケンス図=約 65 図 合計約 120 図

以上のパターンについて、其々の目的と UML 図について以下の 3 点に着目し分類した。

- (1) パターンの目的
- (2) 静的な構造=クラス図の構造
- (3) 動的な振舞 =シーケンス図にある振舞の構造

2.1. 目的によるパターンの分類

設計者がパターンを選択する時、適用する意図やシステムのレイヤー・アーキテクチャ等から適切なものを選ぶ。しかし大抵は J2EE 等のアーキテクチャを中心に分けられており、

設計者の意図が中心とした選択ができない。

そこで設計者の意図を実現できる全パターンを俯瞰するため、レイヤー・アーキテクチャを超えて目的毎にまとめた。抜粋元である書籍の記述を参考にして分類すると、以下 6 つの傾向がある。

- サービスのインターフェイス化
- サービスやエンティティの仲介者
- 構造 (レイヤー) の分離
- データやオブジェクトに関連
- CRUD を実現
- ネットワークを介した処理 (分散処理、並列処理、メッセージング等)

これらを更に詳しく分析し、表 1 のように分類した。アーキテクチャにより同じ目的を持つ傾向がある。しかしアーキテクチャだけでは割り切れない部分がある。これは「サービスのインターフェイス化」に多く見られた。

表 1 目的によるパターンの分類

目的		パターン	目的	パターン
複雑なサービスの簡単なインターフェイス	窓口を一つに限定	Broker[1], Façade[3], Web Service Broker (Custom XML Messaging 戦略, Java Binder 戦略[4], (EJB, JMS Queue, JMS Topic) Service Locator[4][5], Application Façade[6], Message Façade[6], Session Façade[6]	データアクセス	オブジェクトに保持 Data Access Object[4][5], Lazy Load[4], Read Only RowSet[4], RowSet Wrapper List[4], Domain Store (Custom Persistence 戦略[4], JDBC for Reading[6]
	内部の複雑さを見せない	Command[3], Business Delegate (Delegate Adapter 戦略) [4][5], EJB Command[6]	コマンドでアクセス	Data Access Command Bean[6]
	窓口でレイヤーを一つに見せる	Application Service (Layer 戦略, Command 戦略)[4]	データの一時保存	View Handler[1], Data Transfer Object[4][5], Composite Transfer Object[4][5], Value List Handler[4], Data Transfer HashMap[6]
	オブジェクトへ間接アクセス	Proxy (Invocation Handler 使用)[3]	オブジェクトへ間接アクセス	Proxy (Invocation Handler 使用)[3]
	サービスを非同期呼出	Service Adapter (Aggregator 戦略)[4]	エンティティを共有	Flyweight[3]
サービスの共通インターフェイス	窓口を一つに限定	Singleton[3], Iterator[3]	エンティティを集約	Chain of Responsibility[3], Composite[3], Composite Entity (Store Optimization 戦略)[4]
	最終的に 1 つのインターフェイス	Decorator[3]	生成	Factory が作成 Abstract Factory[3], Factory Method[3], (D)IO Assembler[4], Business Delegate Factory[5], DAO Factory [4] [5], DTO Factory [6]
	共通部分をまとめる	Template Method[3], Observer[3], Chain of Responsibility[3]		その他 Builder[3], Prototype[3]
	振舞を一つ	State[3], Strategy[3]	検索	Value List Handler[4]

	に見せる			
仲介者	オブジェクトの仲介者	Mediator[3]	更新	Observer[3], Memento[3]
	サービスの仲介者	Adapter (=Wrapper)[3], Service Adapter[4], Web Worker マイクロアーキテクチャ(Action Adapter, Work Adapter)[4]	分散処理・並行設計	Broker [1], 反応体[7], 先行体[7], 受取人・接続人 [7], オブジェクト復旧[7], オブジェクト同期体[7], 非同期完了型トークン[7], 半同期・半非同期[7], ボディガード[7]
作業の構造を組織化		Command Processor[1], Master-Slave[1]	並列処理	Transaction Context[6], Version Number[6], 楽観的並行処理[5], 悲観的並行処理[5]
クラスと実装の分離		Bridge(=Handle)[3], Visitor[3], Business Object[4]	メッセージング	Point to Point Distribution[5], Competing Consumers[5], メッセージング[5], Control Bus [5], Sequenced Message[5], Pipes and Filters[4][5], Contents-Based Routing [5]
MVCの分離		MVC[1], PAC[1]	その他	Interpreter[3], Object-Relational Mapping[5]

*J2EE パターンや EJB パターンなどの種類があるが、本稿は一元化して記述している。

*POSA=[1] GoF デザインパターン=[3] J2EE パターン…[4][5] EJB パターン=[6] その他=[7]

2.2. 静的構造によるパターンの分類

実装時の静的構造を表すのは、クラス図のクラス構造である。各パターンのクラス図を眺めると、クラス構造の特徴は大きく 3 種類に分類できる。

- 汎化型 (インターフェイス化)
…抽象クラスを利用するパターン
- 移譲型 (集約・コンポジット)
…移譲や集約を用いるパターン
- その他 (汎化/移譲なし、複合型)
…汎化も移譲も使わないパターンか、汎化と移譲が複雑に組み合わせるパターン
「汎化」とは継承関係があるか否かで、「移譲」とは構造に集約・コンポジットがあ

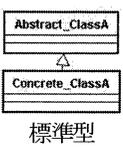
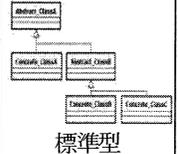
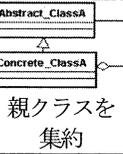
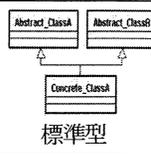
るか否かである。自分自身を移譲する構造は、特に「再帰構造」と定義した。

汎化があり、更に移譲がある構造も数多くあった。この場合、汎化内部の集約・コンポジットという項目にした。移譲の形は、親子のどちらかが移譲される構造に分けられたからである。まず汎化で分けた後、更に移譲で分けるとすっきり分類できた (表 2)。

「その他」には汎化も移譲も無い構造や、クラスが一つだけで関連性が無い構造が入る。もしくは汎化と移譲が複雑に絡み合い、汎化や集約に分類できなかったものを入れた。

静的構造をより詳しく分類した結果が表 2 である。

表 2 静的構造によるパターンの分類

構造		パターン	構造		パターン
単純な汎化	 <p>標準型</p>	Layer[1], Template Method[3], Chain of Responsibility[3], Proxy[3], Application Service - Application Service Layer 戦略[4], Transfer Object[4][5], Business Delegate[4][5], O-R Mapping[5], Session Façade[6], Null オブジェクト[7]	汎化の多重構造	 <p>標準型</p>	Decorator[3], Service Façade[3], 拡張オブジェクト[7], 役割オブジェクト[7]
	 <p>親クラスを集約</p>	Flyweight[3], Command[3], State[3], Strategy[3], Builder [3]	子クラスが親クラスを複数継承	 <p>標準型</p>	Adapter[3]

	<p>親クラスを コンポジット</p>	Command Processor[1], View Handler[1]	単純な 移譲	<p>標準型</p>	Master-Slave[1], Whole-Part[1], Memento [3]
	<p>子クラスを集 約</p>	Composite[3], Interpreter[3]	再帰 構造	<p>標準型</p>	Business Object [4], Composite Entity [4], TOAssembler[4]
双子 の 汎 化	<p>標準型</p>	Mediator[3], Iterator[3], Abstract Factory [3], Factory Method [3], Business Delegate Factory[5], DAO Factory[4][5]	その他	<p>複雑な複合体</p>	Value List Handler[4], ボディ ガードパターン[7], 接続人 パターン[7], 先行体パタ ーン[7], 反応体パターン [7], オブジェクト同期体 パターン[7], (カスタ マイズ可能)オブジェクト復 旧パターン[7]
	<p>親クラスを 集約</p>	Bridge[3], Observer[3]		<p>汎化も移譲も なし</p>	Data Access Object[4][5], Procedure Access Object[4][5], Service Locator[4][5], Application Service[4], 通信/メッセージングパ ターン[5], Essence[7], 型オブ ジェクト[7]
	<p>親クラスを コンポジット</p>	Visitor[3]			

*POSA=[1] GoF デザインパターン=[3] J2EE パターン…[4][5] EJB パターン=[6] その他=[7]

2.3. 動的振舞によるパターンの分類

パターンの動的振舞を表すのは、シーケンス図にあるメッセージ群である。シーケンス図を分析した結果、動的振舞においても複数のパターンに共通する構造があると判断できた。振舞による分類結果が表3である。

静的構造と違う点は、目的が共通すると振舞も似た構造になるものが在ったことである。しかし一概にそうとは言えず、目的は全く別でも同じ動的振舞を持つものもあった。

静的構造とは違い、メッセージ等が完全に一致するパターンは無かった。そこで振舞の大きな構造に少しでも共通点があるものを分類した。

例えば DTO[4][5]や、DTO Factory[6]、Cache[3]や Flyweight[3]などの目的は全く別である。しかし Factory 等の共通ライフラインがあり、かつ「生成、Set、Get、更新」という一連のメッセージ構造が似ていた為、同じ構造とした。

表3 動的振舞によるパターンの分類

振舞		パターン	など
サービ スに1 ヶの窓 口(サー ビスの 集約)	Service Locator系	Service Locator[4][5], EJB Service Locator[4][5], JDBC Service Locator[4][5], JMS Service Locator[4][5]	Group A Adapter[4], Visitor[3], Singleton[3], Memento[3], Mediator[3]
	Service Activator系	Service Activator (Service Activator Aggregator 戦略)[4]	Command系 Command[3], EJB Command[6]
	Group A	Master-Slave[1], Whole-Part[1], Strategy[3], State[3], Message Façade[6], Session Façade[6]	その他 Point-to-Point Distribution[5] Generic Attribute Access[6] Client Dispatcher Server[7] Forwarder Receiver[7]
仲介者	Adapter系	Adapter[3], Action Adapter[4], Service Adapter[4], Work	データ アクセス Group A Data Access Object (Data Access RowSet[4], RowSet Wrapper List 戦略)[4], Value List Handler[4]

ス	Lazy Load[5]
	Group B Domain Store[4]
エンティティの集約	Composite Entity[4], Composite Entity (Store Optimization 戦略)[4]
生成など	Builder[3], Prototype[3], TO Assembler[4], Statefull Business Delegate Factory[5], (D)TO Factory[4][5], Generic Attribute Access[6], Essence[7]
Create/Set/Get/Update有(=TO系)	Cache[3], Flyweight[3], Value List Handler[4], DTO[4][5], Data Access Command Bean[6], Data Transfer HashMap[6], DTO Factory[6]
Group A	Proxy[3], Decorator[3], Business Object[4]

*POSA=[1] GoF デザインパターン=[3] J2EE パターン…[4][5] EJB パターン=[6] その他=[7]

(目的混合)	Application Service (Layer 戦略, Command 戦略)[4], Web Service Broker (Custom XML Messaging 戦略) [4], Business Delegate (Invocation Handler 戦略, Delegate Adapter 戦略) [4][5]
その他	Command Processor[1]
	View Handler[1]
	Observer[3]
	Intercepting Filter[7]
	Null Object[7]

3. パターン定義言語の提案

3.1. パターン定義言語の構造と特徴

静的構造と動的振舞は、各図の共通点から分類した。この事から各パターンの構造には、共通部分があると判断した。

パターン定義言語にはこの共通部分を利用して、重複する部分の記入を省ける形式にした。各パターンの類似性が分かりやすいという利点もある。

具体的には、複数パターンに共通する構造を土台として定義する(図1)。そして個々のパターンは土台の共通部分と呼び出ししながら、特有な箇所を定義する。

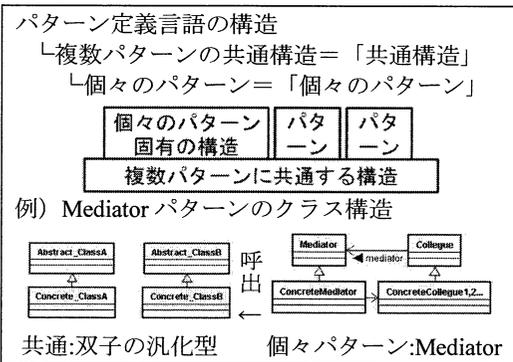


図1 パターン定義言語の構造

この2層構造は、クラス定義と振舞定義の両方に共通する。ただし他のパターンと共通部分がないものは、例外的に土台は無しとする。

記述形式はXMLである。XMI形式と「図の構造を定義する」という共通点があるため、属性などを参考にした。また直感的に分かりやすい構造にする為、XML Schema等のタグも参考にした。その結果リスト1の構造を定義

した。以下に注意が必要なタグを説明する。

- <Diagram>...パターンは静的構造と動的振舞の2図を定義する。静的構造=クラス図、動的振舞=シーケンス図である。
- <Pattern>...共通構造(土台)と個々のパターンを定義するタグ。クラス図の場合とシーケンス図の場合で、中の要素が異なる。
- <GroupX>...人が図を見るときに分かりやすい、クラスやメッセージのまとめ(図2)

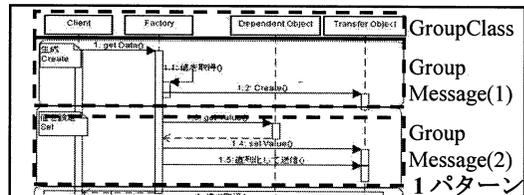
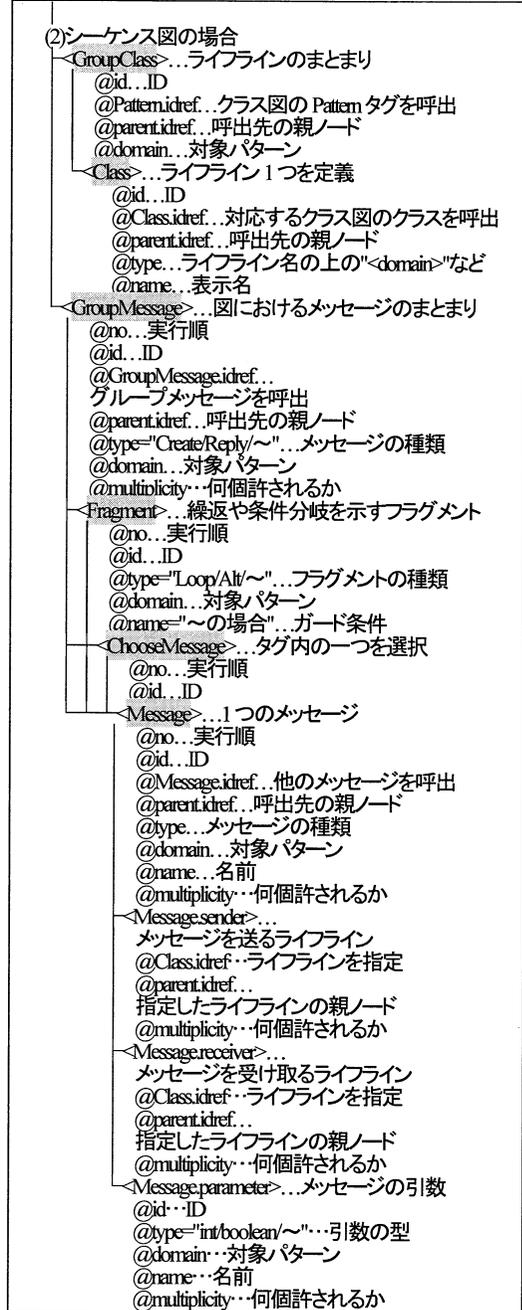
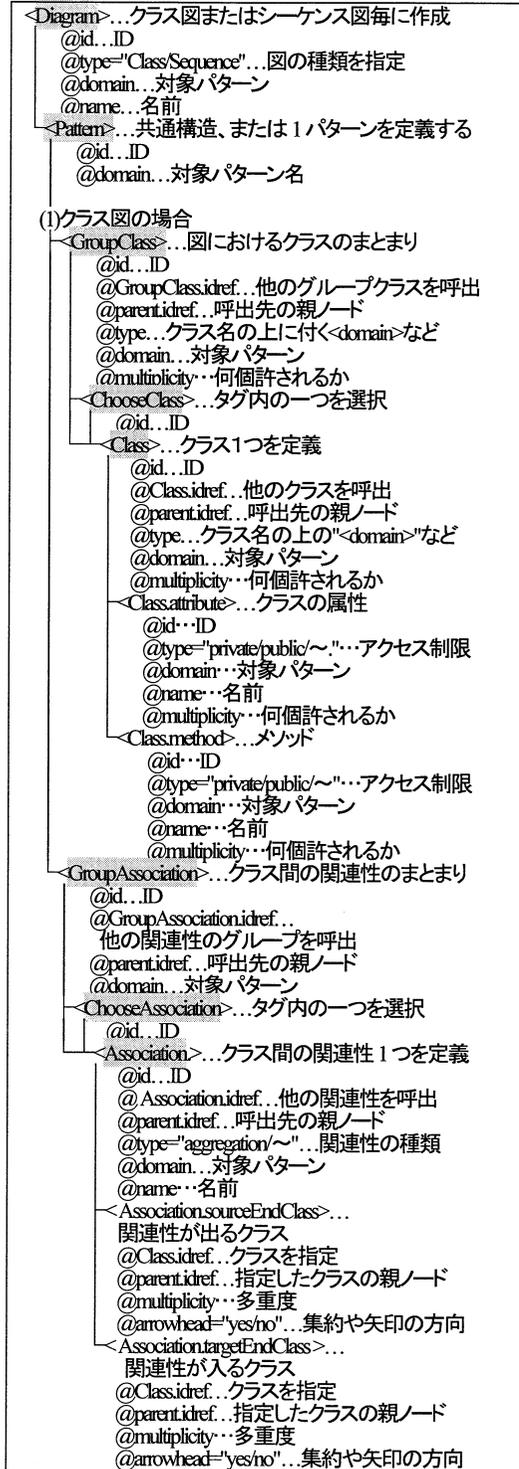


図2 グループ定義の例(シーケンス図の場合)

- <ChooseX>...タグ内から一つの要素だけ選ぶ。選択基準は、各要素の属性 domain (対象パターン) が合致するもの。
- <Class>,<Association>,<Message>...各図の要素=クラス、関連性、メッセージ。クラス1つにつき<Class>タグ1つを作成し、属性とメソッドをタグ内に定義する。他の関連性(Association)やメッセージも同様である。シーケンス図のライフラインは<Class>タグで表現する。
- <Fragment>...シーケンス図のフラグメントを表現する。繰り返しや条件分岐に使用する。
- 属性 X.idref... 共通構造(土台)と、個々のパターンに共通部分の目印として使用する。呼び出す際に名前など独自の要素は上書きして利用する。

リスト 1 パターン定義言語の構造



4. パターン定義言語の記述実験と課題

収集した約 120 のシーケンス図とクラス図のうち、88 図について、定義言語で記述し実験した。その結果、以下の課題が見つかった。

- (a) パターンをテキスト形式で記述してい

くと、定義しにくい。タグの整合性が取りにくく、多数ある属性の必要・不必要が分からない。

- (b) パターンの共通構造から呼び出す要素は、@parent.idref と @Class.idref / @Message.idref 等で指定する。しかし大きな区切りである<Pattern>や<Diagram>タグを超えて呼び出すと、呼出先の特定が難しい。
- (c) シーケンス図のライフライン<Class>タグが、クラス図のクラスの<Class>タグと重複して紛らわしい。
- (d) パターンを定義するうえで不必要な属性がある。

5. パターン定義言語の修正

前章の問題点を解決する為、以下の修正を加えてパターン定義言語を再定義した。

- (1) (a)を解決するため、Excel でパターンの定義を補助するファイルを作成した
- (2) (a)を解決するため、区別するメリットがないタグを統合した。(XML Schema の定義方式を採用)

- <GroupClass>,<GroupAssociation>,<GroupMessage>タグ
 - →<Group>タグに統合
- <ChooseClass>,<ChooseAssociation>,<ChooseMessage>タグ
 - →<Choice>タグに統合

※ <Group>,<Choice> タグ内には、<Class>,<Association>,<Lifeline>,<Message>のいずれか1種しか入らない。

但し<Group>タグ内に<Group>,<Choice>を入れるのは可。

- (3) (b)を解決するため、呼出先を指定する属性(Class.idref 等)と、呼出先の親ノードの id 属性 (parent.idref) に、呼出元のノードツリーから呼出先が分かれる枝ノードの@idを追加した

属性 parent.idref="

'Pattern@id 属性'. 'Class@id 属性'"

↑呼出元のノードツリーから呼出先が分岐する枝の@id。例は<Diagram>は同じで<Pattern>が分かっている場合

- (4) (c)を解決するため、シーケンス図のライフラインは<Lifeline>タグへ変更した
 - (5) (d)を解決するため、不要な属性を削除し、必要な属性を追加した
- 以上の修正を施した後、再びパターンを定

義した。再定義した記述の一例を、リスト 2 (静的構造)とリスト 3 (動的振舞)に挙げる。背景色付部分が主な変更点である。

リスト2 クラス構造の記述例

```
<Diagram id="TwinInterfaces" type="Class"
domain="FactoryMethod..." name="Class_TwinInterfaces">
<!--共通する構造の定義-->
<Pattern id="TwinInterfacesBase" domain="">
<!--クラスの記述--> [タグ統合]
<Group id="TwinInterfaces" type="" domain=""
multiplicity="">
<Class id="AbstractA" domain="" name="AbstractA"
multiplicity="1"/>...クラスが複数つく...
</Group>
<!--関連性の記述--> [タグ統合]
<Group id="TwinInterfaces" type="" domain=""
multiplicity="">
<Association id="InterfaceA" domain="" name="">
<Association.sourceEndClass Class.idref="AbstractA"
parent.idref="TwinInterfacesBase.TwinInterfaces"
arrowhead="1" multiplicity="1"/>
<Association.targetEndClass Class.idref="ConcreteA"
parent.idref="TwinInterfacesBase.TwinInterfaces"
arrowhead="" multiplicity="*">
</Association>...関連性が複数つく...
</Group>
</Pattern>
<!--個々のノパターンの定義-->
<Pattern id="TwinInte Factory" omain="FactoryMethod...">
<!--クラスの定義--> [タグ統合]
<Group no="" id="TwinInterfaces FactoryClass" type=""
domain="FactoryMethod..." multiplicity="">
<Class id="AbstractFactory" Class.idref="AbstractA"
parent.idref="TwinInterfacesBase.TwinInterfaces"
domain="FactoryMethod..." name="AbstractFactory"
multiplicity="1"/> [情報を追加]
<Class.method id="createProduct()Product" type="+
domain="FactoryMethod..."
name="createProduct()Product" multiplicity="1"/>
<Class.attribute id="SubjectState" type="-"
domain="FactoryMethod..."
name="SubjectState" multiplicity="1"/>
</Class>...クラスが複数つく...
<Choice id="Adapter"> [タグ統合]
<Class id="AbstractFactory" Class.idref="AbstractA">
...クラスの記述がつく...
</Choice> ...<Choice>が複数つく...
</Group>
<!--関連性の定義--> [タグ統合]
<Group id="TwinInterfaces FactoryAssociation" type=""
domain="FactoryMethod..." multiplicity="">
<!--共通構造の関連性を呼出-->
<Group no="" id="BaseSelfComposition"
Group.idref="TwinInterfaces" [情報を追加]
parent.idref="TwinInterfacesBase.TwinInterfacesBase"
type="" domain="" multiplicity="1"/>
...共通構造の関連性の定義が続く...
</Group>
<!--ノパターン独自の関連性を定義-->
<Association id="Request" domain="FactoryMethod..."
name="依頼">
<Association.sourceEndClass Class.idref="Client"
parent.idref="TwinInterfaces FactoryClass"
arrowhead="" multiplicity="1"/>
```

```

<Association.targetEndClass
Class.idref="AbstractFactory"
parent.idref="TwinInterfaces.FactoryClass"
arrowhead="1" multiplicity="1"/>
<Association>...関連性が複数つく...
</Group>
</Pattern>...Pattern タグ(個々のパターン)がつづく...
</Diagram>

```

リスト 3 振舞の記述例

```

<Diagram id="TO" type="Sequence" domain="DTO..."
name="SequenceDiagram TO">
<!--共通する構造の定義-->
<Pattern id="TOBase" domain="DTO..."> タグ統合
<Group no="" id="TOBaseLifeline" type=""
domain="DTO..." multiplicity="">
<Lifeline id="Client" Class.idref="Client" 変更
parent.idref="simpleInterfacePresentation" type=""
name=""/>...ライフラインが複数つく...
</Group>
<!--メッセージの定義--> タグ統合
<Group no="1" id="Create" type="Create" domain=""
multiplicity="1">
<Message no="1" id="ClientRequest" domain=""
name="get Data" multiplicity="1">
<!--ライフラインを参照-->
<Message.sender.Lifeline.idref="Client"
parent.idref="TOBaseLifeline"/>
<Message.receiver.Lifeline.idref="Factory"
parent.idref="TOBaseLifeline"/>
<Message>...メッセージが複数つく...
</Group>
</Pattern>
<!--個々のパターンの定義-->
<Pattern id="TwinInterfaces.Factory" domain="Factory...">
<!--ライフラインの定義--> タグ統合
<Group no="" id="DataTransferHashMapLifeline" type=""
domain="DataTransferHashMap" multiplicity="1">
<Lifeline id="Client" type="" name="Client"/> 変更
...ライフラインが複数つく...
</Group>
<!--メッセージの定義--> タグ統合
<Group no="1" id="TO_DataTransferHashMap" type=""
domain="DataTransferHashMap" multiplicity="1">
<!--共通構造のメッセージを呼出--> タグ統合
<Group no="1" id="FactoryGet" Group.idref="Create"
parent.idref="TO.TOBase" type="" domain="Data.Transfer..."
multiplicity="1"> 情報を追加
<Message no="1" id="ClientRequest" domain="DataTr..."
name="get Entity" multiplicity="1">...</Message>
...共通構造のメッセージの定義が続く...
</Group>...複数つく...
<!--独自のメッセージを定義-->
<Message no="2" id="ReplyClientGetData"
domain="DefT..." name="Reply" multiplicity="1">
<Message.sender.Lifeline.idref="TO"
parent.idref="TOBaseLifeline"/>
<Message.receiver.Lifeline.idref="Client"
parent.idref="TOBaseLifeline"/>
<Message>...メッセージが複数つく...
<Choice id="Select"> タグ統合
...メッセージが複数入る...
</Choice>...複数つく...
</Group>
</Pattern>...Pattern タグ(個々のパターン)が複数つく
</Diagram>

```

6. おわりに

ソフトウェアパターンの利用を支援するツールを開発する為に、POSA、GoF デザインパターン、J2EE パターン、EJB パターン等から約 80 のパターンを収集し、約 120 の UML 図から目的と構造を分析した。

次に分析結果から各パターンの構造に共通部分があり、この類似性が特徴になると判断できた。そこで目的と静的構造、動的振舞の共通度に着目して分類した。最後に共通する構造を利用して各パターンを定義する、定義言語を設計した。

設計した言語で約 90 図を記述し評価したところ、定義が難しいという問題があった。そこで定義言語の構文を修正し、パターンを再定義した。実験データである UML 図の約 30 図は記述されず、定義言語に改善の余地が残った。

再定義したデータは、設計支援ツールで「パターン定義データ」として利用する予定である。このツールは適用できるパターンを案内し、既存のクラス図とシーケンス図に選択された型を組込む機能を想定している。この UML 図に組み込むパターンのデータを作成するために、言語を定義しパターン定義の記述実験を行った。

参考文献

- [1] ソフトウェアアーキテクチャ, Frank Buschmann 他, 近代科学社, 2000
- [2] エンタープライズアプリケーションアーキテクチャパターン, マーチン・ファウラー, 翔泳社, 2005
- [3] デザインパターン改訂版, Erick Gamma 他, ソフトバンククリエイティブ, 1999
- [4] J2EE パターン第 2 版, Deepak Alur 他, 日経 BP, 2005
- [5] J2EE デザインパターン, William Crawford, オライリージャパン, 2004
- [6] EJB デザインパターン, Floyd Marinescu, 日経 BP 社, 2003
- [7] プログラムデザインのためのパターン言語, PLoPD Editors, Softbank Publishing, ソフトバンククリエイティブ, 2001