第9回：『科学的モデリング』継承⑨～「リスコフ置換原理」とタイプ(型)階層

第9回の話題～「リスコフ置換原理」とタイプ(型)階層

第8回に続き「タイプ(型)置換原理(the principle of type substitution／type substitution principle)」の解説です。第7回で「タイプ(型)置換原理」の中で特に有名な「リスコフ置換原理」(文献[9-2])を紹介しました【注9-1】。オブジェクト指向開発では「タイプ(型)置換原理」は継承関係をタイプ(型)安全かつ効果的に利用するために欠かせない最重要原理の１つです。今回も「リスコフ置換原理」を含む「タイプ(型)置換原理」の理解のために解説を続けます。

前回と前々回で「リスコフ置換原理」は、他の「タイプ(型)置換原理」と比較して厳しい「タイプ(型)置換原理」であると紹介しました。今回は「リスコフ置換原理」の意味を理解するための例を用いて「事前条件」を集中的に解説してきます【注9-2】。

今回のテーマは：

- 概念モデル／分析モデルと設計モデルの違い
- 「リスコフ置換原理」と「事前条件」の再定義(override)
- 「リスコフ置換原理」と多相(ポリモフィズム／多態)の関係
- 「リスコフ置換原理」とタイプ(型)階層

です。

【注9-1】
「リスコフ置換原理」は文献[9-2]で詳細に記述されていますが、リスコフ自身は、この論文の中で「リスコフ置換原理」という呼び方をしていません。後に他の研究者によって、この論文の内容を「リスコフ置換原理」と呼ぶようになったようです。

【注9-2】
「タイプ(型)置換原理」は、専門的にはタイプ(型)理論と演繹的な推論を使った原理です。そのため原理の証明を含めて理解するには、タイプ(型)理論と形式論理学の演繹的な証明技法の理解が求められますが、一般の開発エンジニアの方は、「リスコフ置換原理」の意図を理解すれば、まずは十分と言えます。

概念モデル・分析モデルと設計モデルの理論と原理

これまでの技術コラムの中で継承関係を解説するときは、常に「タイプ(型)」の視点から解説してきました。

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC.　ALL RIGHTS RESERVED.

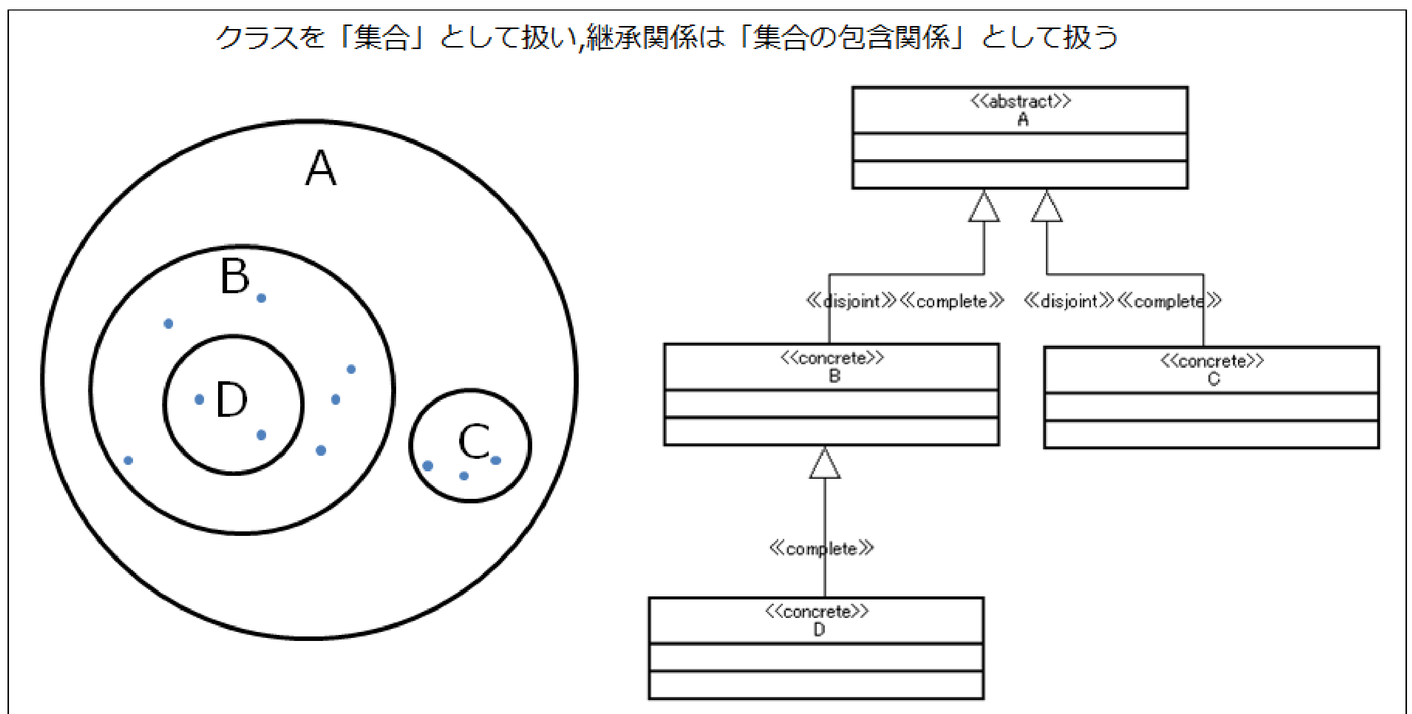
「リスコフ置換原理」もタイプ(型)について述べている原理です(第7回コラム参照)。

ここで、簡単に『科学的モデリング』における概念モデル、分析モデルおよび設計モデルの中でタイプ(型)の扱いについて言及しておきます。

モデルの開発は、開発初期の概念モデルからはじまり分析モデル、設計モデルとモデルを段階的に進化させていきます。

概念モデルと分析モデルでは、クラスは数学の集合として扱います。このときは設計モデル同様に「集合=タイプ(型)」と考えても問題ありませんが、設計モデルよりもタイプ(型)のプロパティ(特性)の記述を詳細に定義しません。また、分析作業中であるからです。

概念モデルと分析モデルでは、継承関係は集合の包含関係として表します【図9-1】。継承関係は集合の包含関係で表現することで、特定のプログラム言語、OSおよび動作環境などの実装を意識することなく対象の概念の把握が可能になります。また、クラスを「集合」、継承関係を「集合の包含関係」、クラス間の関連を「写像」で表現することで数学や論理学の理論や定理を用いて正確なモデルの表現と分析が可能となります。



【図9-1】

一方、実装を意識した設計モデルでは、実装する言語、OSおよび動作環境を意識してクラスおよびクラス間の継承関係をモデルで表現する必要があります。オブジェクト指向プログラミングでは、継承関係は「差分プログラム」

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC. ALL RIGHTS RESERVED.

「多相(ポリモフィズ／多態)」などの機能により生産性や品質を向上させるための技法として継承関係を利用します。

そのため、概念モデル、分析モデルではクラスを集合、継承関係を集合の包含関係で表現することが設計モデルでは困難になります。包含関係は、単に集合のc の順序関係を表現するだけなので、継承関係によるサブクラスタイプ(型)のプロパティ(特性)の「特殊化」「拡張」を表現することはできません[第4回コラム参照]。

以上から開発初期の概念モデル、分析モデルから設計モデルに移行するときに、設計や実装の理論と原理・原則を設計モデルに適用する必要があります。つまり、概念モデル、分析モデルのモデル開発とはモデル開発で利用する「理論」「原理」および「技法」が設計モデルでは変化します。そのため概念モデル、分析モデルから設計モデルに移行するときには注意が必要となります。

そのため、概念モデル、分析モデルとして正しいモデルでも、設計モデルへと進化させていく過程でモデルの正当性と妥当性を維持するためには、概念モデル、分析モデルのクラス図を修正して、設計モデルを作成する必要がでてきます。今回のコラムでは、この点についても関係する話題を扱います。

なお、開発初期の概念モデル、分析モデルの考え方や開発方法については、今後の技術コラムの中で改めて解説します。『科学的モデリング』は、設計モデル同様に開発初期の概念モデル、分析モデルについても非常に正確なモデルを確実かつ迅速に開発できるアプローチを備えています。いずれこのコラムで【概念モデル／分析モデル】編を扱いご紹介できるようにしたいと考えています。

「リスコフ置換原理」が主張すること

「設計モデル」ではクラスを「タイプ(型)」として明確に意識し、プロパティ(特性)を明確に定義する必要がでてきます。特にクラス間に継承関係は、クラスのタイプ(型)を強く意識する必要があります。今回の解説する「リスコフ置換原理」は、スーパータイプ(型)とサブタイプ(型)の「振る舞い(意味的)の整合性の成立」について述べた原理です。

第7回のコラムで紹介した表の１つを再掲載し「リスコフ置換原理」について復習します【表9-1】。

科学的モデリング規則：リスコフ置換原理(Liskov substitution principle)のエッセンス①
● サブクラスのタイプ(型)はそのスーパークラスのタイプ(型)と置換可能(substitutable)でなければならない ☞ 多相(ポリモフィズム／多態)が可能でなければならない ● スーパークラスのタイプ(型)の「クラス不変条件」と全ての操作の「事前条件」「事後条件」を満たすことをサブクラスで保証しなければならない

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC.　ALL RIGHTS RESERVED.

【表9-1】

【表9-1】中の：

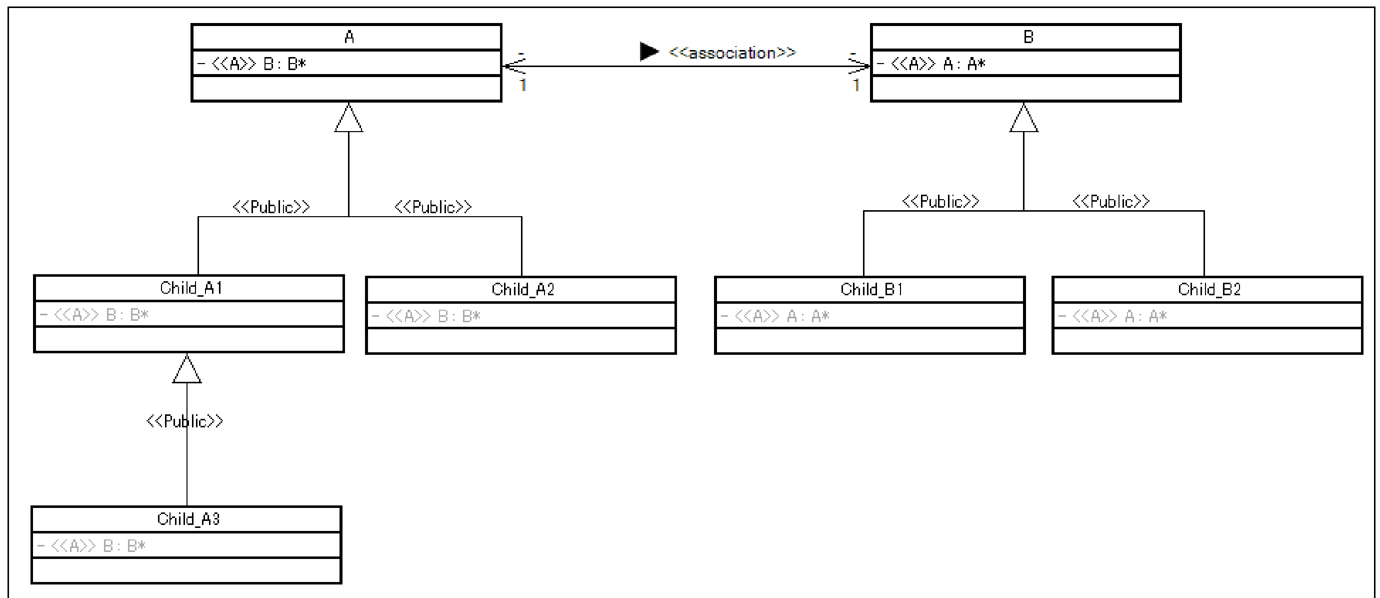
- サブクラスのタイプ(型)はそのスーパークラスのタイプ(型)と置換可能(substitutable)でなければならない（多相(ポリモフィズム／多態)が可能でなければならない）

に注目します。

オブジェクト指向プログラミングおいて、サブクラスのオブジェクト(インスタンス)への参照は、そのスーパークラスのオブジェクト(インスタンス)への参照に代入可能です。そして、このことを踏まえて「リスコフ置換原理」の内容を考えると、一見単純なことを述べている印象を持ちますが、実は厳しいことを主張していることが見えてきます。理由を【図9-2】を使って解説します。

【図9-2】ではクラス「A」とクラス「B」があり、それぞれサブクラスが定義されています。各クラスの属性、操作、各種制約など他のプロパティ(特性)は、ここでの解説に直接関係しないので省略しています。
各クラスには唯一の属性として関連先のクラスの参照を明示的に表示しています。

例えば、クラス「A」には、属性の区画に« A » B:B*と表示がありますが、これは関連先相手のクラス「B」の参照を保持していることをC++言語風に表現しました。なお、« A » は関連先のための属性であることを示しています。



【図9-2】

クラス「A」のサブクラスであるクラス「Child_A1」，クラス「Child_A2」およびクラス「Child_A3」は、継承の理論に基づき自分のスーパークラスのタイプ(型)のプロパティ(特性)を全て継承します。つまり、【図9-2】に示され

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC.　ALL RIGHTS RESERVED.

ているようにクラス「A」が保持する関連先相手のクラス「B」の参照も継承します(サブクラスに継承されているタイプ(型)のプロパティ(特性)を灰色で表示しています)。

そのため、クラス「A」を含めクラス「A」の全ての子孫クラスは、クラス「B」のオブジェクト(インスタンス)との関連を保持できることになります。そして、これはとりもなおさず、クラス「B」を含めクラス「B」の全ての子孫クラスのオブジェクト(インスタンス)との関連を保持できることを意味します。以上のことはクラス「B」の階層についても同様のことが成立します。スーパークラスの参照を属性として持つことで、このスーパークラスの子孫のサブクラスは全て保持できるのですから、特定のクラスの参照に限定されないため高い柔軟性が得られます。多相(ポリモフィズム／多態)のメリットです。

以上のことは、JavaやC++やその他のオブジェクト指向言語でプログラム開発をしたことがある方なら、当然の話であり、特に難しいことではありません。オブジェクト指向言語の構文の文法で認められているからです。

しかし、スーパークラスとサブクラス間の継承関係で、多相(ポリモフィズム／多態)を実現することを前提にする場合は、単にオブジェクト指向言語の構文の文法を満たすだけでは不十分です。「振る舞い(意味的)の整合性」も満足させることが必要となり、この時状況は一変します。

ここで再度、第7回のコラムで紹介した「リスコフ置換原理」について述べたもう１つの表を再掲載します【表9-2】。

科学的モデリング規則：リスコフ置換原理(Liskov substitution principle)のエッセンス②
● サブタイプ(型)の操作の「事前条件(pre-condition)」は、スーパータイプ(型)の操作の「事前条件(pre-condition)」と**条件が「同じ」であるか、条件を「弱める」ことができる**
● サブタイプ(型)の操作の「事後条件(post-condition)」は、スーパータイプ(型)の操作の「事後条件(post-condition)」と**条件が「同じ」であるか、条件を「強める」ことができる**

【表9-2】

【表9-2】において最初の操作の「事前条件」について述べている部分に注目しましょう。

- サブタイプ(型)の操作の「事前条件」はスーパータイプ(型)の「事前条件」と**条件が「同じ」であるか、条件を「弱める」ことができる**

とあります。

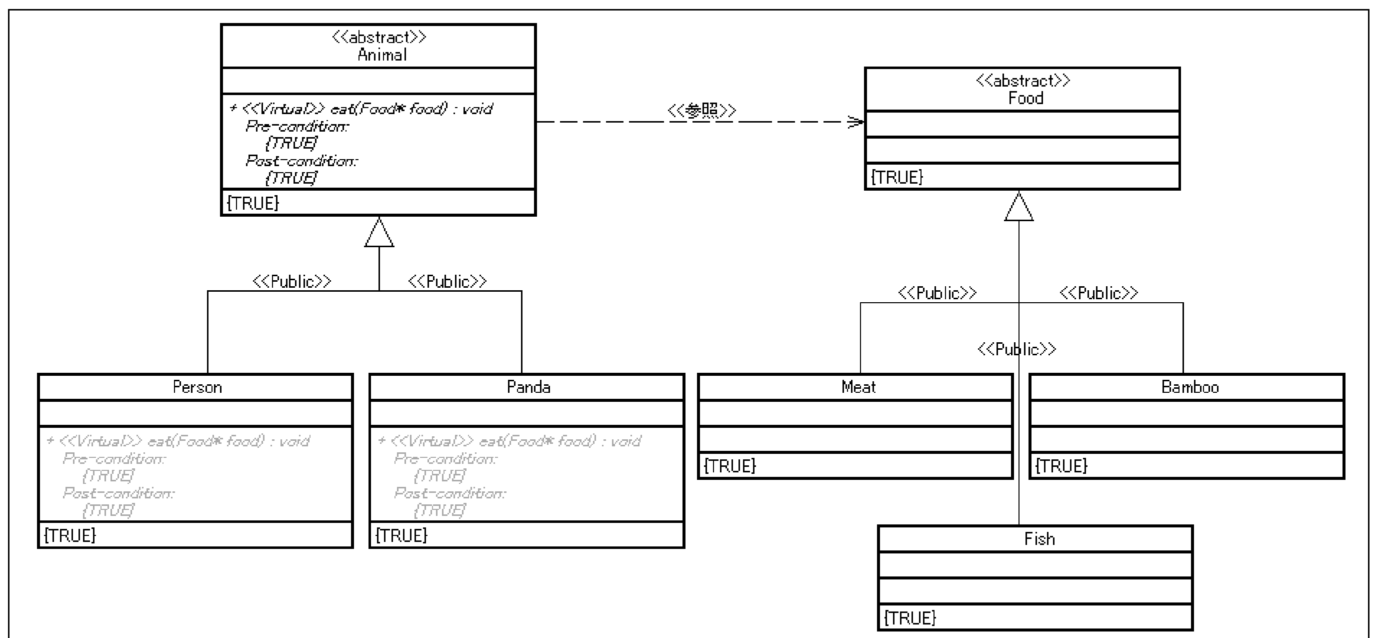
COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC. ALL RIGHTS RESERVED.

なお、サブクラスがスーパークラスから操作を再定義せずそのまま継承した場合は問題ありません。この場合は、サブクラス操作の「事前条件」は、スーパータイプ(型)の「事前条件」と(条件の「強さ」)が「同じ」になります。このとき、サブクラスのタイプ(型)とスーパークラスタイプ(型)は「同型」です[第4回コラム参照]。

注意が必要なのはサブクラスがスーパークラスから継承した操作を再定義(override)した場合です。【表9-2】ではスーパータイプ(型)とサブタイプ(型)の「振る舞い(意味的)の整合性の成立」を満たすには、『操作の「事前条件」が**「同じ」**であるか、**「弱める」**ことができる』とありますが、このことを満足することは、しばしば困難な場合があります。例を用いて説明します。

「リスコフ置換原理」が破られるとき

【図9-3】は動物と食べ物について表現したクラス図です。



【図9-3】

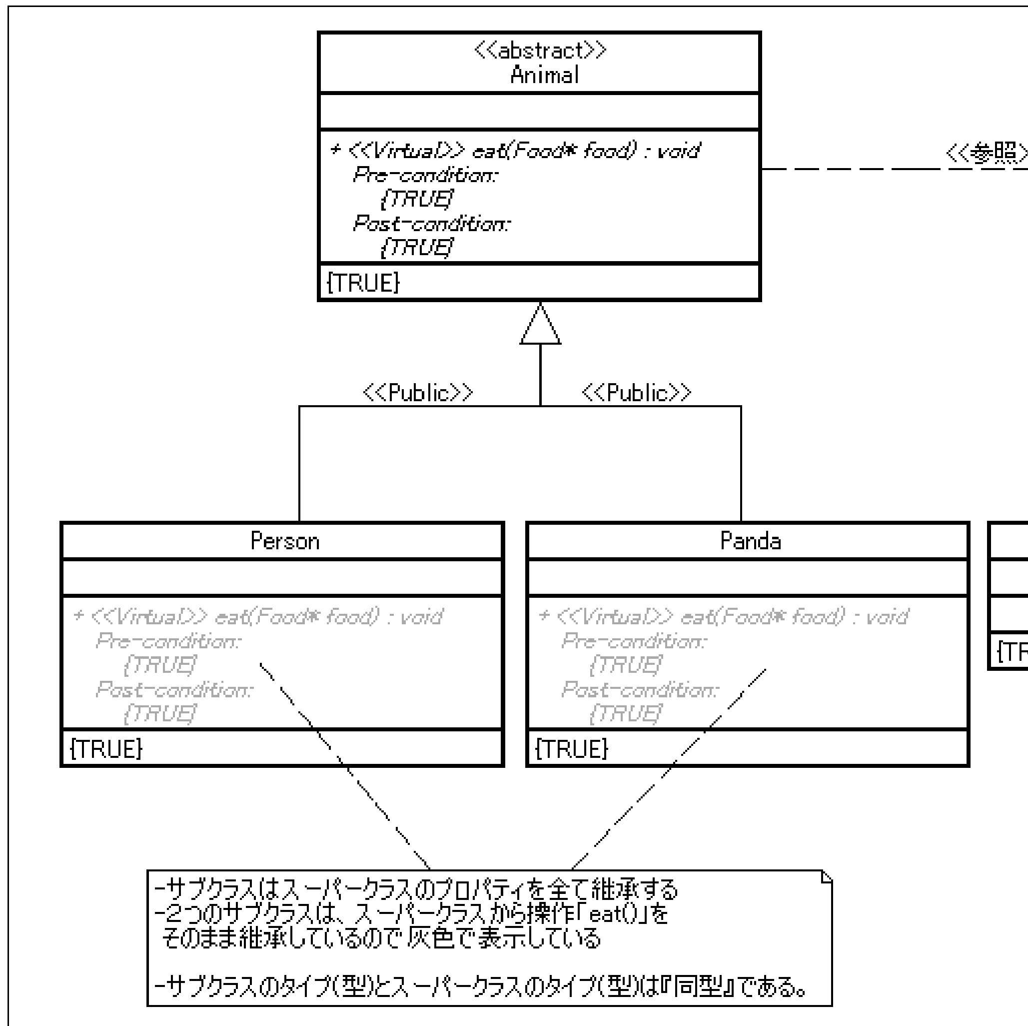
【図9-3】の左側はクラス「Animal」とそのサブクラス「Person」とサブクラス「Panda」が定義されています。右側は、クラス「Food」とそのサブクラス「Meat」、サブクラス「Fish」およびサブクラス「Bamboo」が定義されています。さらにクラス「Animal」からクラス「Food」に向けて「依存関係« 参照» 」が存在しています。

ここで、【図9-3】の左側のクラス「Animal」とそのサブクラス「Person」とサブクラス「Panda」に注目してみます【図9-4】。クラス「Animal」は仮想である操作「eat(Food* Food) void」が定義されています。引数にクラス

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC.　ALL RIGHTS RESERVED.

「Food」への参照を持つので、クラス「Food」のタイプ(型)を参照する必要があります。そのため「依存関係« 参照 » 」が引かれています。

さて、【図9-3】のクラス図が意味することは特に説明がなくても分かると思います。このクラス図は、クラス「Animal」、クラス「Person」およびクラス「Panda」が食べる食料を表しています。



【図9-4】

サブクラス「Person」とサブクラス「Panda」を見ると、スーパークラス「Animal」から操作「eat(Food* Food) void」を再定義(override)しないでそのまま継承しています【図9-4】。なお、サブクラス「Person」とサブクラス「Panda」のプロパティ(特性)が灰色での表示は、スーパークラス「Animal」から継承しているプロパティ(特性)を省略せず明示的に示していることを表しています。

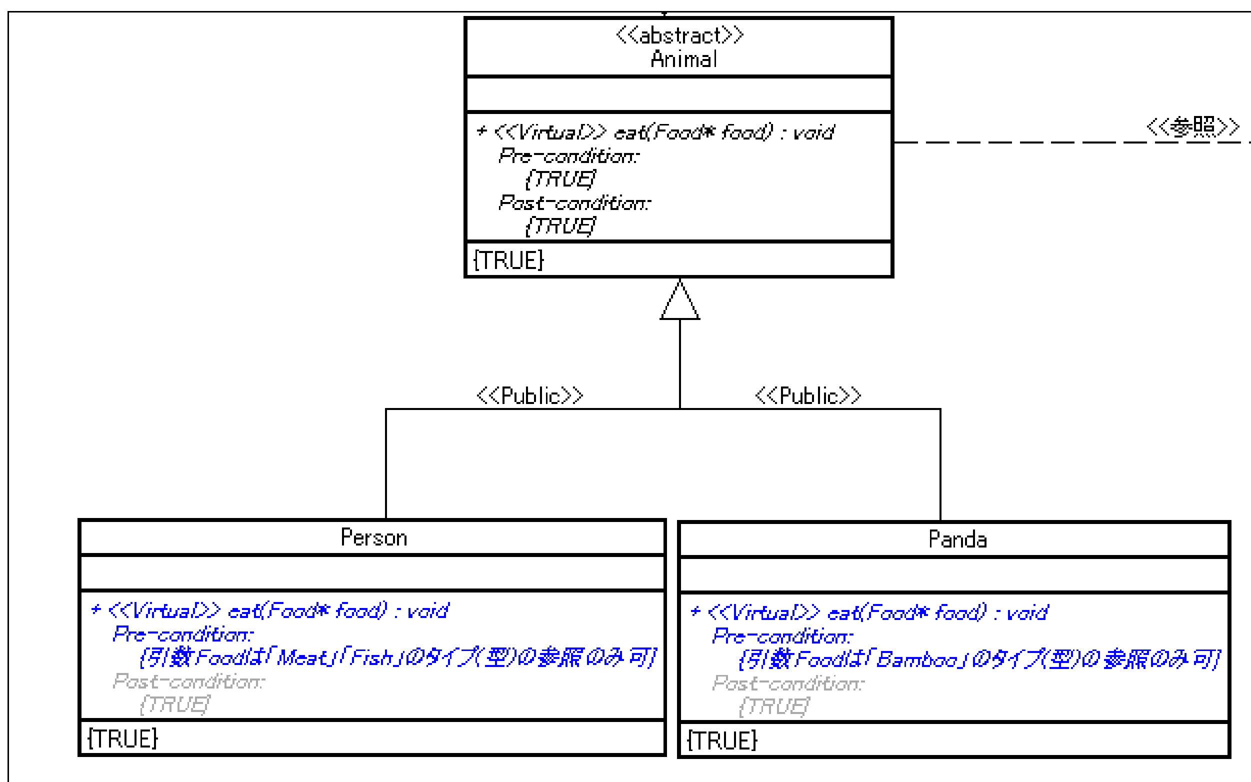
COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC.　ALL RIGHTS RESERVED.

ここからが重要な点です。

クラス「Person」が継承した操作「eat(Food* Food) void」の引数のタイプ(型)は、クラス「Food」のタイプ(型)への参照です。上記したようにスーパークラスの参照には、このスーパークラスの全子孫となるサブクラスのオブジェクト(インスタンス)への参照を代入できるのですから、クラス「Person」の操作「eat(Food* Food) void」の引数には、クラス「Meat」、クラス「Fish」およびクラス「Bamboo」のオブジェクト(インスタンス)への参照が代入できます(できてしまいます)。

人間は肉や魚は食しますが、竹は食べないので、モデルを開発したモデラーは、クラス「Person」の操作「eat(Food* Food) void」の引数には、クラス「Bamboo」の参照を代入することは意図していないはずです。

クラス「Panda」も同様です。クラス「Panda」の操作「eat(Food* Food) void」の引数には、クラス「Bamboo」のオブジェクト(インスタンス)への参照しか代入できないのが「意味的に正しい」筈です。

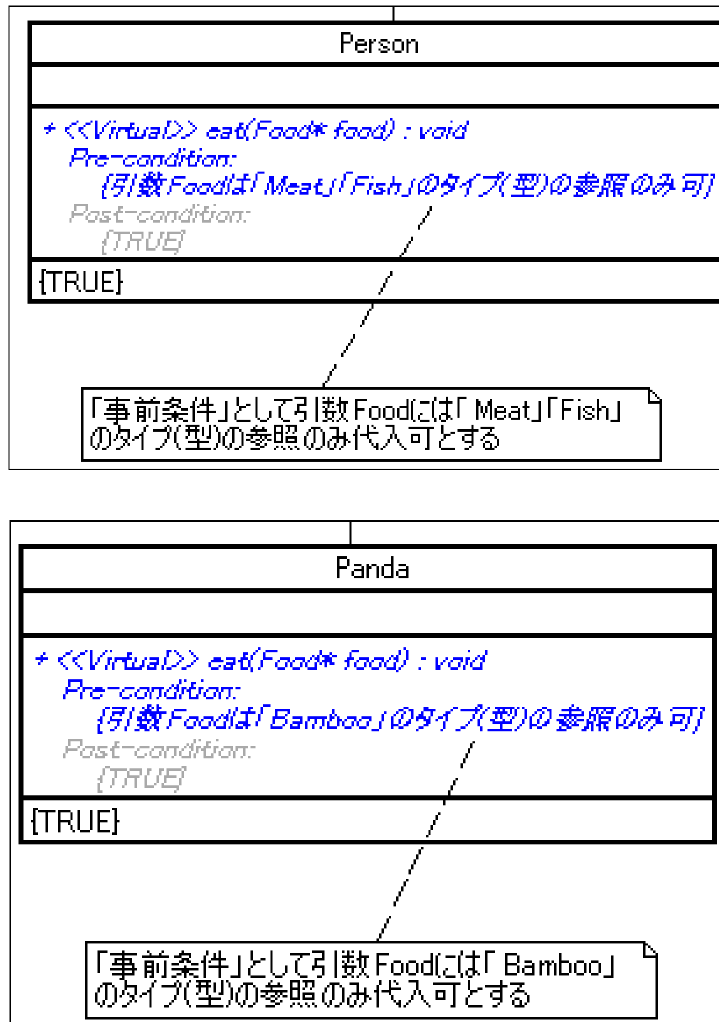


【図9-5】

そこで、【図9-5】のようにクラス「Person」とクラス「Panda」の操作「eat(Food* Food) void」を正しく呼び出すための条件を「事前条件」に追加します。クラス「Person」とクラス「Panda」の操作「eat(Food* Food)

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC.　ALL RIGHTS RESERVED.

void」は、適切な食糧のみ対応する操作として再定義(override)されました。青色で表示は再定義(override)されたことを示しています【図9-5】【図9-6】。



【図9-6】

【図9-5】【図9-6】のように、クラス「Person」とクラス「Panda」の操作「eat(Food* Food) void」を、それぞれのクラスに適切な食糧のみ対応する操作として再定義(override)することは、各クラスの振る舞いの「意味的な正しさ」を満たします。

ところが、今度はスーパークラスとサブクラスの多相(ポリモフィズム／多態)が不成立になります。

スーパークラス「Animal」のプロパティ(特性)を再度確認してみます。スーパークラス「Animal」の操作「eat(Food* Food) void」には何も「事前条件」が設定されていません。つまりスーパークラス「Animal」の操作「eat(Food* Food) void」のFoodの参照型と持つ引数Foodに代入に対する制約が無いことを意味します。

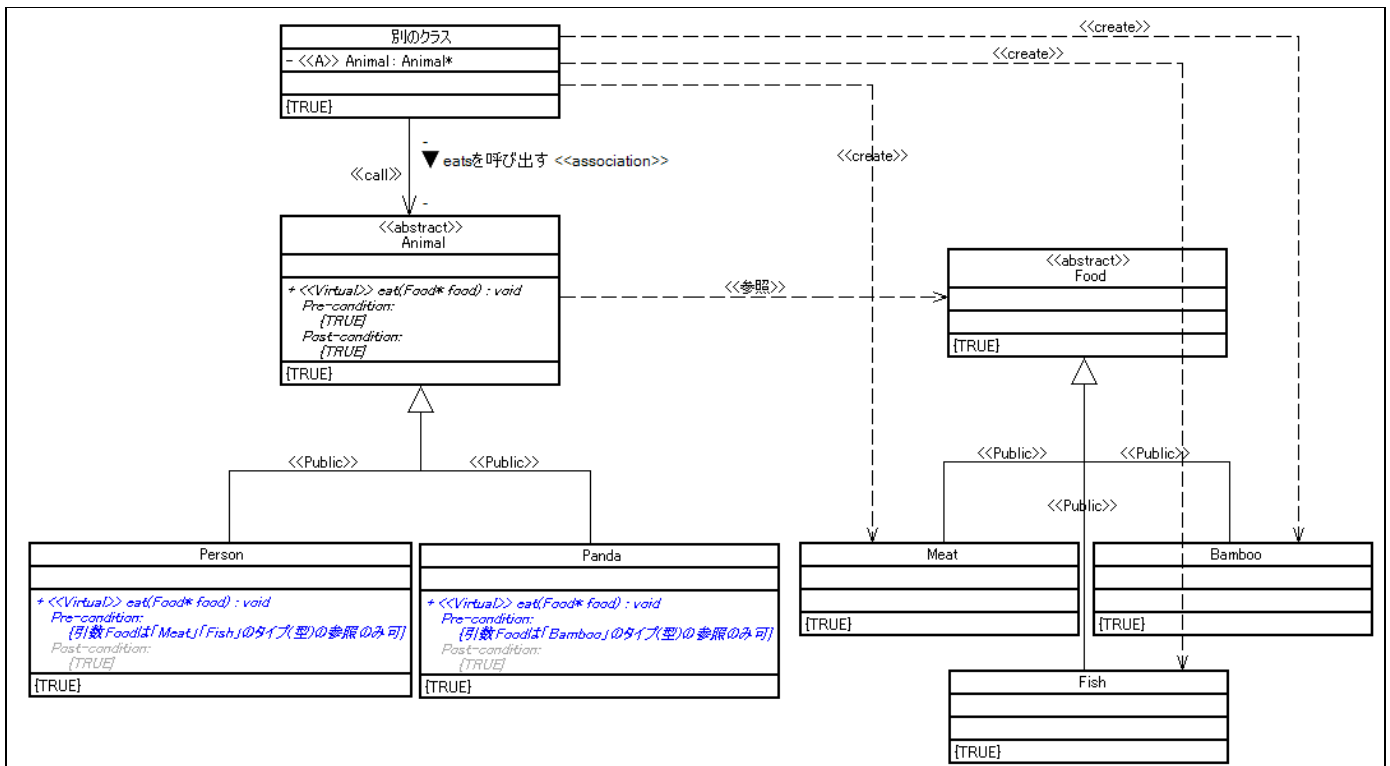
COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC.　ALL RIGHTS RESERVED.

スーパークラス「Animal」の操作「eat(Food* Food) void」は、外部に公開する操作であるため、この操作を正しく呼び出す制約である「事前条件」に何も条件が無いことを宣言(表明)していることになります。

この操作を呼び出す制約は無いということですから、無条件に呼び出せることになります。

スーパークラス「Animal」は抽象クラスです。オブジェクト(インスタンス)を生成できないので、多相(ポリモフィズム／多態)の実現を目的として定義されたクラスです。別の表現を使えば、インタフェースとして機能することを意図して定義されたクラスです。そのため、このインタフェースを通じてサブクラスのオブジェクト(インスタンス)にアクセスする他のクラスのオブジェクト(インスタンス)は、サブクラスについてのプロパティ(特性)を知らずに呼び出すことになります。つまり、インタフェースのクラスの操作に何も「事前条件」が無いというにも関わらず、実際に呼び出されるサブクラスの操作には条件が設定されているために問題が生じます。

クラス図にクラス「別のクラス」と関連および依存関係を追加した図が【図9-7】です。クラス「別のクラス」のオブジェクトは、クラス「Meat」、クラス「Fish」およびクラス「Bamboo」のオブジェクト(インスタンス)を生成します(そのため依存関係« create» がクラス「別のクラス」からクラス「Meat」、クラス「Fish」およびクラス「Bamboo」に存在しています)。



【図9-7】

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC.　ALL RIGHTS RESERVED.

なお、クラス「別のクラス」のオブジェクトは、既にクラス「Person」とクラス「Panda」のオブジェクト(インスタンス)と関連が既に設定されているとします（誰がクラス「別のクラス」、クラス「Person」とクラス「Panda」のオブジェクト(インスタンス)を生成したかについては、ここでの解説に関係しませんので割愛しています）。

クラス「別のクラス」のオブジェクトは、インタフェースとして機能するクラス「Animal」の参照を通じて、クラス「Person」とクラス「Panda」のオブジェクト(インスタンス)の操作「eat(Food* Food) void」を呼び出します。

このとき、クラス「別のクラス」のオブジェクトは、操作「eat(Food* Food) void」の引数Foodに、自分が生成したクラス「Meat」、クラス「Fish」およびクラス「Bamboo」のオブジェクト(インスタンス)のいずれの参照を渡すことが自由と考えています。インタフェースとして機能するクラス「Animal」では、操作「eat(Food* Food) void」に何も「事前条件」として制約が設定されていないからです。

当然これは問題が発生します。クラス「Person」とクラス「Panda」は、適切な食糧のみ対応するように操作「eat(Food* Food) void」のように再定義(override)たからです。これは、もはやクラス「Animal」の参照にクラス「Person」とクラス「Panda」のオブジェクト(インスタンス)を代入できない、つまり「置換不可能」であることを意味します。

ここで、改めて「リスコフ置換原理」について考察します。【表9-1】と【表9-2】において、ここまでの解説に関連する部分を抽出しました【表9-3】：

科学的モデリングの規則：「リスコフ置換原理」のエッセンス③
● サブクラスのタイプ(型)はそのスーパークラスのタイプ(型)と**置換可能(substitutable)**でなければならない ☞ 多相(ポリモフィズム／多態)が可能でなければならない ● スーパークラスの**タイプ(型)の「クラス不変条件」と全ての操作の「事前条件」「事後条件」を満たすことをサブクラスで保証**しなければならない ● 『サブタイプ(型)の操作の事前条件はスーパータイプ(型)の事前条件と**条件が「同じ」であるか、条件を「弱める」ことができる**』

【表9-3】

ここまで例を使って検討してきたことは、「リスコフ置換原理」に明記されています。サブクラスのタイプ(型)はそのスーパークラスのタイプ(型)と置換可能(substitutable)でなければならない、そして、そのためにはスーパ

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC. ALL RIGHTS RESERVED.

ークラスのタイプ(型)の「クラス不変条件」と全ての操作の「事前条件」「事後条件」を満たすことをサブクラスで保証しなければならないというものです。

- 「スーパークラスのタイプ(型)の全ての操作の「事前条件」を満たすことをサブクラスで保証しなければならない」

とは、サブクラスはスーパークラスから継承する操作において、「事前条件」制約を減らす(緩くする／弱くする)ことはできても、増やす(強くする／厳しくする)ことはできないとうことです

【図9-5】では、クラス「Person」とクラス「Panda」の操作「eat(Food* Food) void」は、適切な食糧のみ対応する操作になるように「事前条件」の条件を追加（強く／厳しく）して再定義(override)しました。「振る舞い(意味的)の整合性」が成立していない訳です。これが、「リスコフ置換原理」に違反した原因です。

さて、それでは【図 9-5】～【図 9-7】のモデルをどのようにすれば、「リスコフ置換原理」に違反しないモデルになるのかという疑問があります。実は根本的な解決方法はありません。間接的な方法によるいくつか回避策はありますが、いずれも方法もモデルの修正が必要となり、かつ、あまりエレガントな方法がありません【注 9-3】。

【注9-3】
この例のケースではモデルの修正を少なく済む方法として、「ダブルディスパッチ(マルチディスパッチとも呼ばれます)」というメカニズムを方法が良く用いられます。

「リスコフ置換原則」と「事前条件」の再定義(override)

クラス「Person」とクラス「Panda」の操作「eat(Food* Food) void」に対して、適切な食糧のみ対応するために「事前条件」の条件を追加（強く／厳しく）して再定義(override)したことにより、「リスコフ置換原理」に違反しました。このことを別の例を用いて考察してみます。

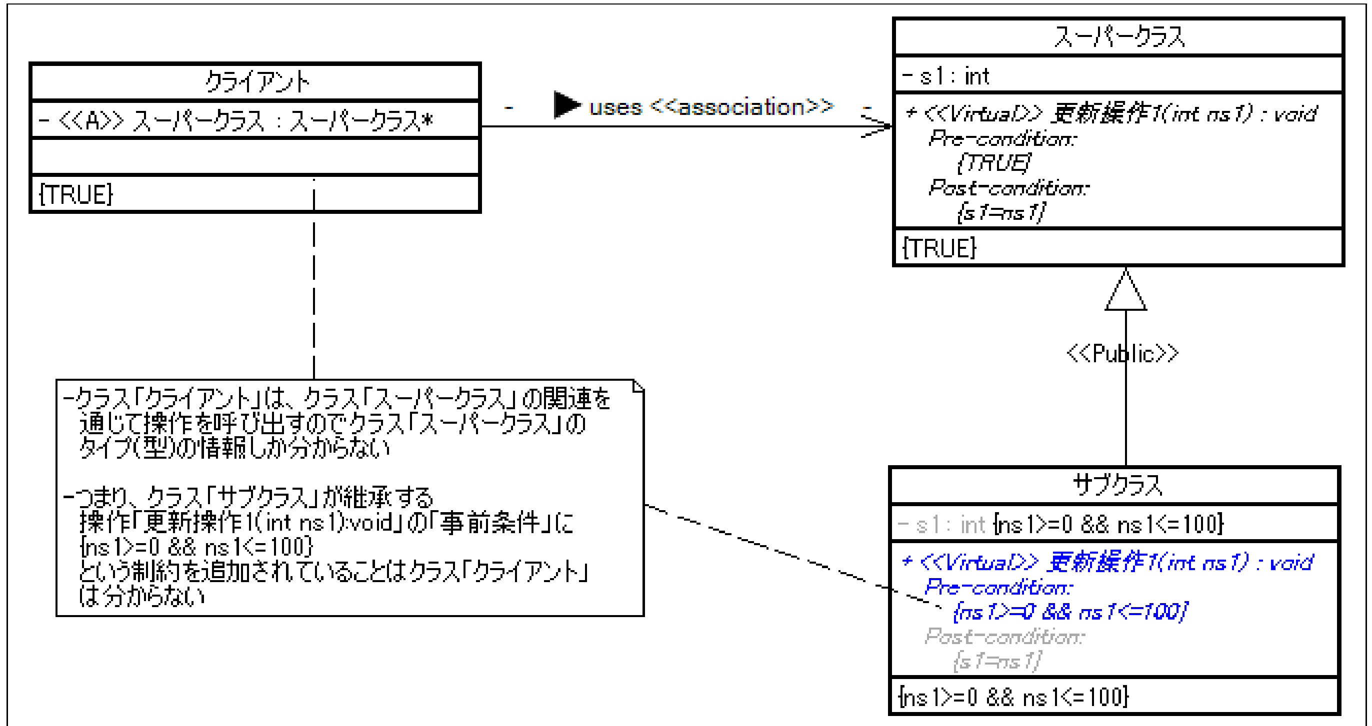
【図9-7】において、クラス「クライアント」は、クラス「スーパークラス」の関連を持っています。クラス「サブクラス」は、クラス「スーパークラス」から継承した「更新操作1(int ns1):void」の「事前条件」に{ns1>=0 && ns1<=100} を追加して再定義(override)しています(「リスコフ置換原理」違反)。

そして、クラス「クライアント」は、多相(ポリモフィズム／多態)により、クラス「スーパークラス」の参照を通じてクラス「サブクラス」のオブジェクト(インスタンス)の操作を呼び出だします。

多相(ポリモフィズム／多態)の注意点は、クラス「クライアント」は、クラス「スーパークラス」のタイプ(型)の情報しか分からないと言う点です。インタフェースであるクラス「スーパークラス」のタイプ(型)の情報が全ての情

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC. ALL RIGHTS RESERVED.

報であるということです。つまり、クラス「サブクラス」が継承する操作「更新操作1(int ns1):void」の「事前条件」に{ns1>=0 && ns1<=100} という制約を追加されていることはクラス「クライアント」は分からない(見えない)訳です。



【図9-8】

「リスコフ置換原理」を満たす難しさの理由は【図9-9】を見る事で分かります。通常、サブクラスはスーパークラスよりも対象の範囲が狭くなります。このことから、サブクラスのオブジェクト(インスタンス)は、スーパークラスのオブジェクト(インスタンス)よりも特殊化された性質を持ちます。これはサブクラスの属性は、スーパークラスの属性よりも条件が厳しく(狭く／増加)なる傾向があるということです。このことを「クラス不変条件」に照らしてみると：

- サブタイプの「クラス不変条件」は、スーパータイプの「クラス不変条件」と「同じ」か「厳しく(狭く／増加)」できる

に該当しています。スーパータイプの「クラス不変条件」と、サブタイプの「クラス不変条件」の整合性につては、第8回のコラムで「論理空間」の考え方を紹介し、サブタイプの「論理空間」は、スーパータイプの「論理空間」内に収まらなければならないと解説しました[第8回コラム参照]。

次に、【図9-9】から特殊化された性質のサブクラスに対して適切にアクセスする操作は、スーパークラスの操作よりも限定されて利用されるべきことは自明です。本来サブクラスはサブクラスの性質(つまりプロパティ(特性))より

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC.　ALL RIGHTS RESERVED.

も、限定される傾向がある訳ですから、スーパークラスの操作をそのまま無条件にサブクラスに適用できることの方が少ない筈です。操作の「事前条件」に何らかの条件を追加することが必要となることが多い筈です。

ところが、多相(ポリモフィズム／多態)を実現するために「リスコフ置換原理」が主張することは：

- サブタイプの操作の「事前条件」はスーパータイプの「事前条件」と条件が「同じ」であるか、条件を「弱める」ことができる

です。

【図 9-9】のモデルは「概念モデル」から見れば正しいモデルと言えます。抽象的ですが現実世界をモデルとしてかなり忠実に写像しているからです。そのためとても直観的です。しかし、このクラス図はプログラムにした時は正しく動作しません。つまり「設計モデル」としては間違いになります。

「設計モデル」は、実世界の概念を忠実に反映することを目的としたモデルではありません。いくら直観的に納得できたしても意味がありません。

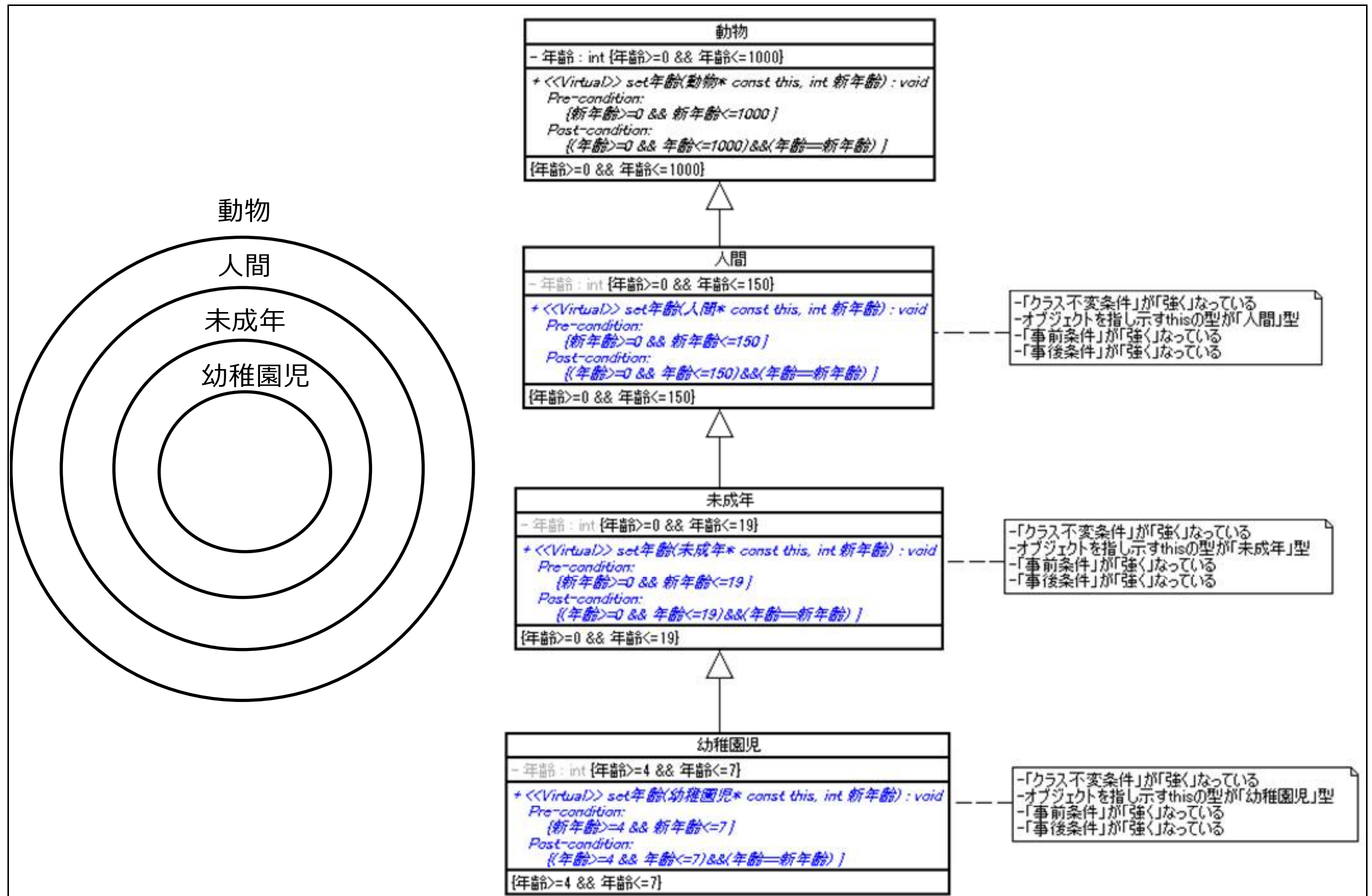
「設計モデル」は、実装言語の仕組みを考慮し、オブジェクト指向およびタイプ(型)理論などの公理や原理を適用して、「正当性のあるモデル」「妥当性のあるモデル」「堅牢なモデル」および「保守・再利用可能なモデル」を作成することを目的としています。

今回のコラムで「リスコフ置換原理」を解説してあらためて気づかされるのは、「設計モデル」の作成や検証は、直観的に正しく見ても何の保証にもならないということです。そして、コンピューター科学に立脚しモデルを作成、検証することが必要であり、品質や生産性の向上に早道であるということです。

科学的モデリング規則：直観に依存せずコンピューター科学に立脚しモデルを作成＆検証する
● 「概念モデル」「分析モデル」「設計モデル」では適用する理論・原理が異なる
● クラスのタイプ(型)の「クラス不変条件」および各操作の「事前条件」「事後条件」により「表明」を定義する
● 設計モデルでは、「表明」を満たすように常に注意する
● 設計モデルでは、サブクラスのタイプ(型)がスーパークラスのタイプ(型)の「振る舞いサブタイプ」が成立するように「リスコフ置換原理」を適用する

【表9-4】

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC. ALL RIGHTS RESERVED.



【図9-9】

「リスコフ置換原理」の適用にはいくつか補足する点があります。

「リスコフ置換原理」を限定的に満たせば問題の無い多相(ポリモフィズム／多態)の利用をする継承関係も実はあります（しかし、この場合はモデルの可変部分と固定部分が明確に分離できる場合に限ります。クラスは開発中および開発後も維持管理や機能追加で修正される可能性があります。モデルの可変部分と固定部分が明確に分離できない場合は、「リスコフ置換原理」に遵守することが重要となります）。

「リスコフ置換原理」を完全に満たさなければ、モデルの「意味的整合性」を保証できないケースは、現実の場面で常にある訳ではありません。継承関係を利用したモデルがタイプ(安全)であり、かつある程度柔軟な設計モデルを構築することは、「リスコフ置換原理」を完全には満たさなくても可能なケースがあるのです。

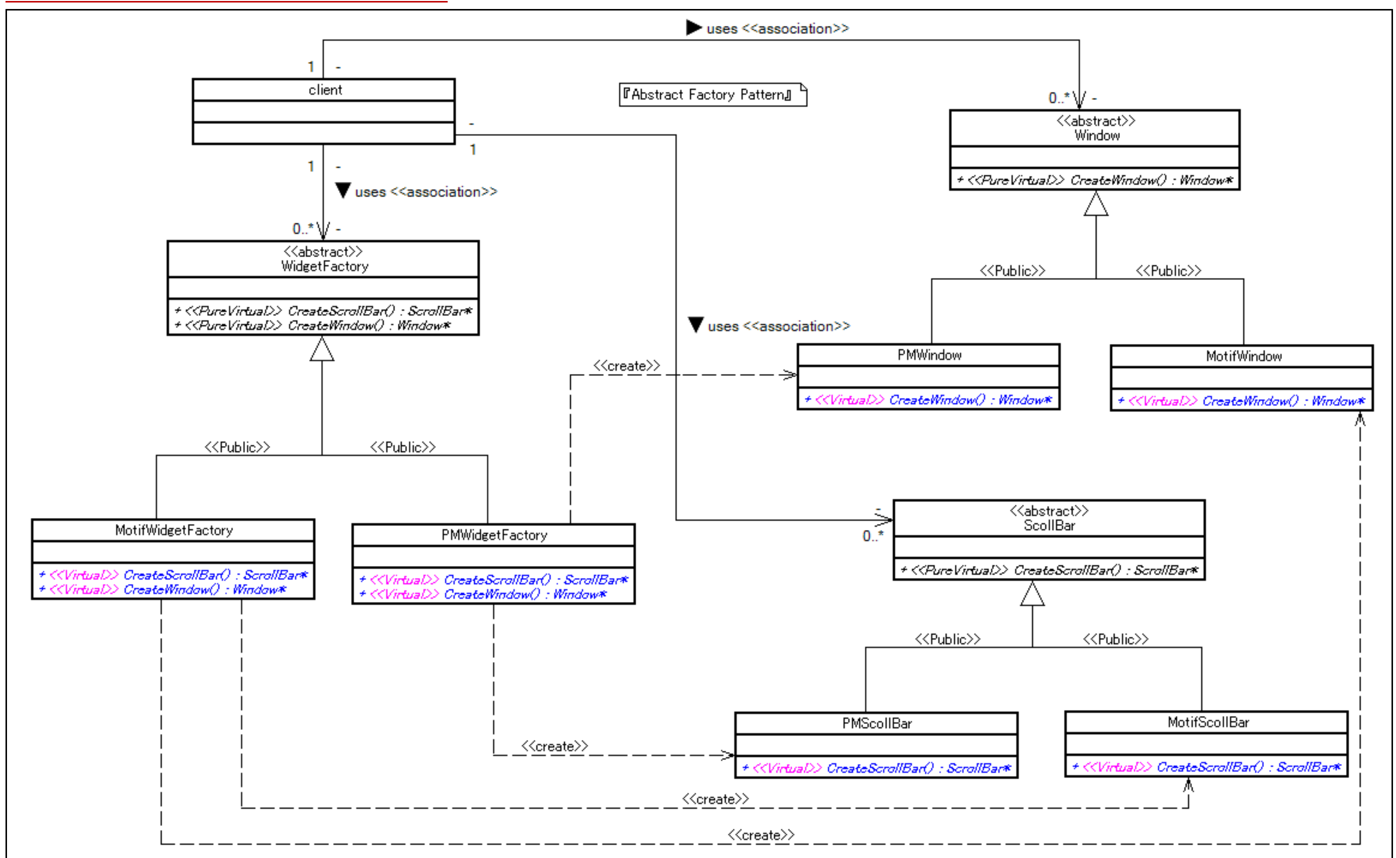
例えば、E.ガンマ達が発表したデザインパターン23個のパターン[文献9-3]では、実開発に利用できる実践的なパターンが紹介されています。パターンの多くは継承関係による多相(ポリモフィズム／多態)を、効果的に使用してい

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC.　ALL RIGHTS RESERVED.

るパターンです。しかし、この継承関係による多相(ポリモフィズム／多態)を利用したパターンの多くは、完全には「リスコフ置換原理」を満たしておらず、厳密には違反しています。

【図9-10】は「Abstract Factory Pattern」と呼ばれる良く知られたパターンの１つです(文献[9-3])。このパターンはAbstract Factoryとして機能するクラス「WedgetFactory」の継承階層中のクラス「PMWindowsFactory」は、対称的に位置する２つの継承階層中のクラス「PMWindows」とクラス「PMScollBar」とのみ、同様にサブクラス「MotifWindowsFactory」は、クラス「MotifWindows」とクラス「MotifScollBar」と”のみ”と関係付けられるように意図されたモデルになっています。そして、この構造がこの「Abstract Factory Pattern」の特徴であり、効果的に機能するポイントになっています。

しかし、これは「リスコフ置換原理」には完全には準拠していません。「リスコフ置換原理」を完全に満足するモデルであれば、スーパークラス同士が関連で結合されている場合には、そのサブクラスの全組み合わせがいかなるコンテキストにおいても、「置換可能」となることを保証する必要があります。



【図9-10】

結論として、「リスコフ置換原理」をどこまで遵守すべきかについては、開発者であるモデラーの意図や状況に依存します。これはより正確に言えば、「リスコフ置換原理」を完全に満たすことを求められたモデルあるか否かとい

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC.　ALL RIGHTS RESERVED.

うことです。つまりモデルに対するニーズと言えます。「リスコフ置換原理」を完全には満たさなくても可能な場合とはどのような場合かについては、今後のコラムで解説していきます。

「リスコフ置換原理」の威力

最後に「リスコフ置換原理」に完全に準拠すると効果が大きいケースを紹介します。「リスコフ置換原理」を完全に満たす多相(ポリモフィズム／多態)のモデルが難しいことや、完全に満たすことが求められない時もあると述べました。それでは、「リスコフ置換原理」は意識しなくて良いのでしょうか？そうではありません。「リスコフ置換原理」を完全に満たすモデルを開発できるなら、それにこしたことはありません。

「リスコフ置換原理」は、継承関係のモデルについての有るべき姿について述べたものではなく、「意味的整合性」を持つ「振る舞いサブタイプ(型)」についての原理です。「振る舞いサブタイプ(型)」が、いかなるコンテキストでも成立する条件について記述しています。

そのため、いかなるコンテキストにおいても「振る舞いサブタイプ(型)」が成立することを、開発者が作成するモデルにどこまで求めるかということを明確にすることが重要です。

いかなるコンテキストでも「振る舞いサブタイプ(型)」が成立する条件を要求されるものとして、クラス階層によるライブラリがあります。

継承関係は階層が深くなればなるほど一般的には効果が大きくなる傾向になりますが、一方で継承関係も「意味的整合性」を保証することが困難になってきます。モデルは開発中、開発後も色々な理由から修正がかけられます。モデル開発者以外のモデラーがモデルの修正や維持を行うことも必要となってきます。あるクラスの属性や関数を１つ修正しただけで、継承階層全体に思いもよらない影響を及ぼすことがあります。

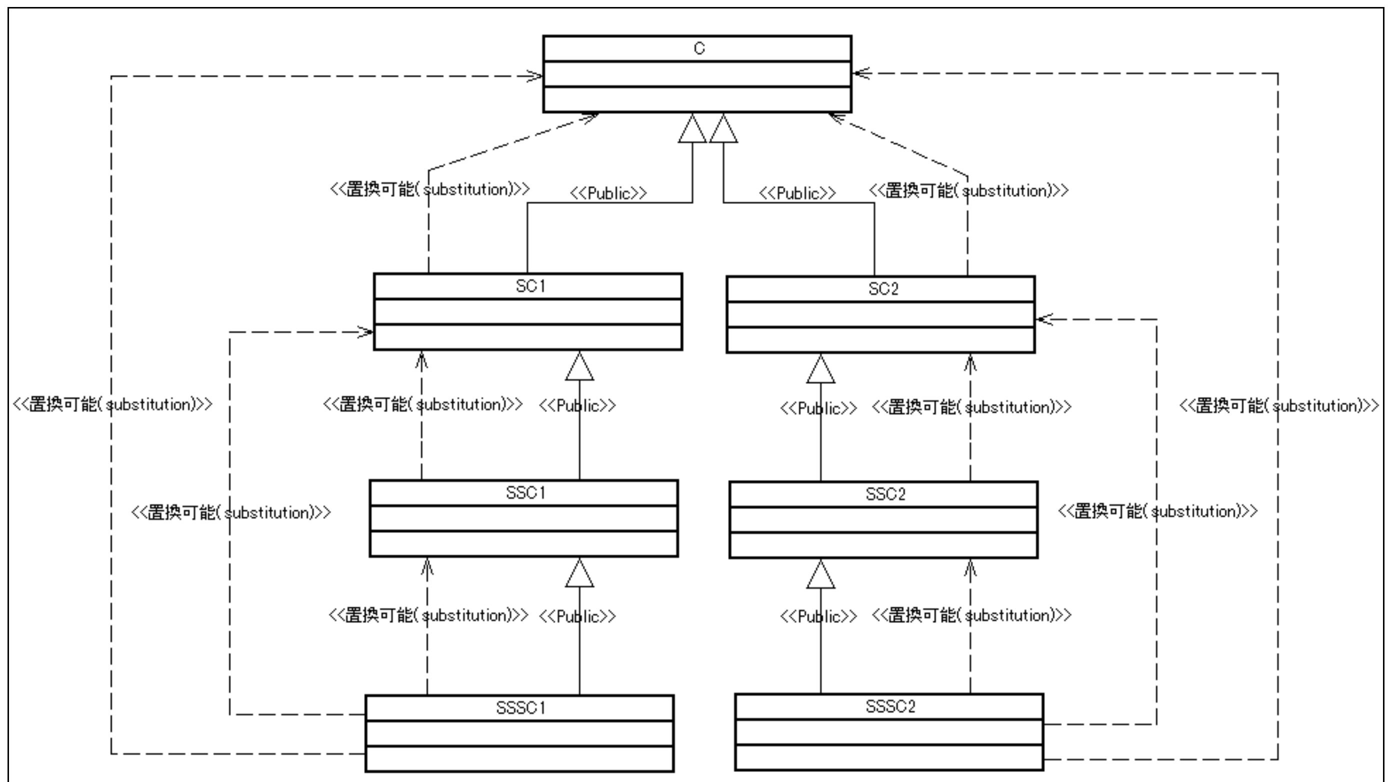
継承階層の「意味的整合性」を確実に保証するためには、やはり「リスコフ置換原則」のような「タイプ置換原理」を適用して首尾一貫した規則を適用しモデルを開発することが必要となってきます。

【図 9-11】を見てください（各クラスは属性や操作などのプロパティ(特性)が省略されています。）

継承階層の「意味的整合性」は、継承階層がどんなに深くなっても保証しなければなりません。「リスコフ置換原理」は継承階層の中で推移的に適用されます。継承階層において「リスコフ置換原理」を満たす場合には「意味的整合性」は完全に保証されます。継承階層の中のクラスを修正・追加・削除した後でも「リスコフ置換原則」を満たす継承階層であれば、決して「意味的な整合性」が破壊される事はありません。

「リスコフ置換原理」を尊重する事で健全なクラス階層を設計する上で重要になります。「リスコフ置換原則」のような首位一貫した原理を用いることは、クラス階層の開発・維持、および多数の開発者が関わる大規模開発では大変有意義となります。

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC. ALL RIGHTS RESERVED.



【図9-11】

「リスコフ置換原理」の形式的表現

（＊ここからは興味がある方だけお読みください）

「リスコフ置換原則」に趣旨を数学的(論理学的)な表記で記述できます。

第8回のコラムで制約条件の比較を行うために数学(論理学)の**「必要条件(necessary condition)」「十分条件(sufficient condition)」「必要十分条件(necessary and sufficient condition)」**について紹介しました。

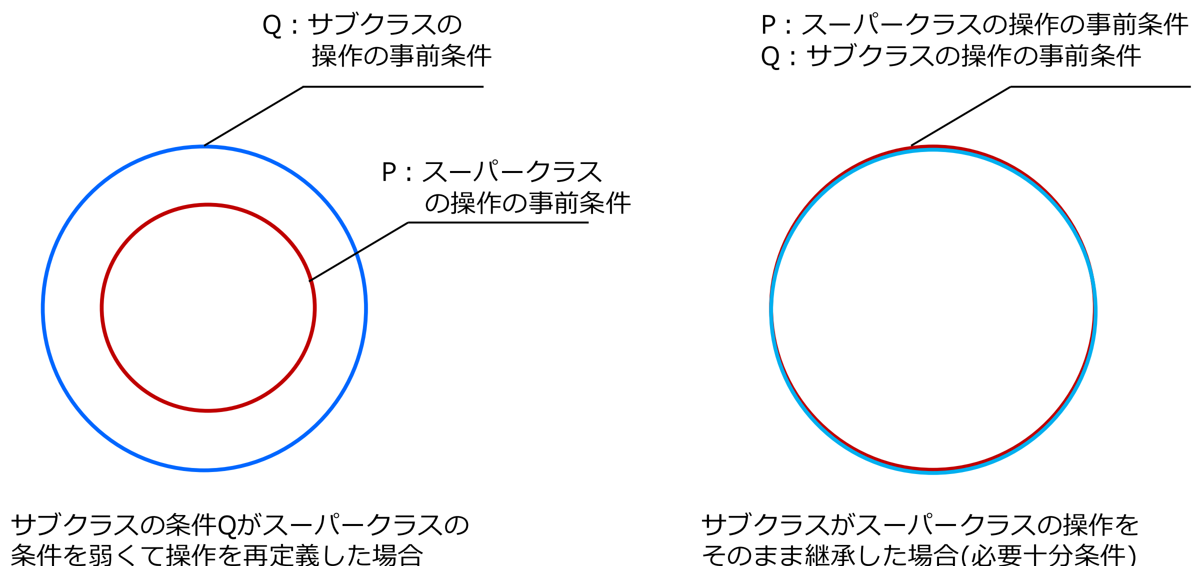
【図9-11】の中の継承階層を「リスコフ置換原則」の「事前条件」と「事後条件」の規則は、数学(論理学)の表現を用いて記述すると【表9-5】のように記述可能です。

科学的モデリング規則：リスコフタイプ置換原理(Liskov type substitution principle)を数学的に表現する
● 『サブタイプ(型)の操作の「事前条件」はスーパータイプ(型)の操作の「事前条件」と条件が**「同じ」**であるか、条件を**「弱める」**ことができる』という意味は： ☞ 『スーパータイプ(型)の操作の「事前条件」が満たされれば、サブタイプ(型)の「事前条件」が必ず満たされる』ということと等価です

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC.　ALL RIGHTS RESERVED.

- ☞ つまり『サブタイプ(型)の「事前条件」は、サブクラスの「事前条件」の**十分条件(sufficient condition)**でなければばならない』と言えます

- 「リスコフ置換原則」が要求する「事前条件」の制約は、スーパークラスの「事前条件」をPと表現し、サブクラスの「事前条件」をQと表現すると**P⇒Q**と記述できます
- この十分条件(sufficient condition)は、継承階層に推移的に成立するので、十分条件(sufficient condition)の**記号(⇒)**を用いて表現すると【図 9-11】の中の継承階層について下記のように記述できます：
  - ☞ $C_{pre_condition} \Rightarrow SC1_{pre_condition} \Rightarrow SSC1_{pre_condition} \Rightarrow SSSC1_{pre_condition}$
  - ☞ $C_{pre_condition} \Rightarrow SC2_{pre_condition} \Rightarrow SSC2_{pre_condition} \Rightarrow SSSC2_{pre_condition}$
- (今回は「事後条件」について触れませんでしたが)「リスコフ置換原理」により【図 9-11】の中の継承階層について下記のように記述できます：
  - ☞ $SSSC1_{post_condition} \Rightarrow SSC1_{post_condition} \Rightarrow SC1_{post_condition} \Rightarrow C_{post_condition}$
  - ☞ $SSSC2_{post_condition} \Rightarrow SSC2_{post_condition} \Rightarrow SC2_{post_condition} \Rightarrow C_{post_condition}$
- 図を用いて表現すると下記の様になります



【表9-5】

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC.　ALL RIGHTS RESERVED.

まとめ＆次回

今回のコラムのまとめとして、解説した重要事項を『科学的モデリング規則』として【表9-6】にまとめておきます。

科学的モデリング規則：「リスコフ置換原則」を適用しモデリングせよ
● 「サブクラス」と「サブタイプ」は明確に異なる概念である
● 継承階層は「クラス」階層ではなく「タイプ(型)」階層と捉えてモデリングする
● 継承階層全体の「意味的整合性」を確実に保証するためには、「リスコフ置換原則」を適用してモデルを開発する
● サブクラスのタイプ(型)が、スーパークラスのタイプ(型)の「振る舞いサブタイプ(behavior subtype)」になるように、「リスコフ置換原理」を継承階層(「タイプ(型)」階層)に推移的に適用する
● 継承階層の中のクラスを修正・追加・削除した後でも「リスコフ置換原則」を満たす場合は、継承階層の「意味的整合性」は完全に保証される
● クラス階層の開発や維持、および多数の開発者が関わる大規模開発では、「リスコフ置換原則」のような首位一貫した原理を適用する
● 「リスコフ置換原則」に遵守しない継承階層を持つモデルを開発するときは慎重に検討する

【表9-6】

次回は、実践的な色合いを強くして継承関係を解説していきます。

参考文献

- 文献[9-1] [BARBARAH. LISKOV 1994] A Behavioral Notion of Subtyping
- 文献[9-2] [Barbara H. Liskov, Jeannette M. Wing 1999]Behavioral Subtyping Using Invariants and Constraints
- 文献[9-3] [Design Patterns: Elements of Reusable Object-Oriented Software 1995](邦訳「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」)

COPYRIGHT©2014～2015 HASHIMOTO SOFTWARE CONSULTING INTERNATIONAL INC.　ALL RIGHTS RESERVED.