

第6回：『科学的モデリング』継承⑥～振る舞い整合性

第6回の話題～首尾一貫した継承の設計基準

第5回のコラムでは「継承パラドックス(inheritance paradox)」を紹介しました。

第6回は**「継承パラドックス(inheritance paradox)」**をもう少し詳しく考察していきます。
そして、継承パラドックスを通じて継承関係を設計するときに何が重要となるのかを考え、科学的な視点から継承関係を設計するときの首尾一貫した設計基準を紹介します。

科学的な視点からの継承関係の「正当性(correctness)」を判断する考え方の土台は、継承関係にあるクラス間の**「振る舞い整合性」**です。今回はこの「振る舞い整合性」を探求していきます。

今回のテーマは：

- **『科学的モデリング』による首尾一貫した「継承の正当性の基準」と「振る舞い整合性」**

です。

『科学的』に継承を理解する

感覚的な方法からの脱却と『科学的アプローチ』への移行

第5回のコラムではクラスの２つの性質である「モジュール」と「タイプ(型)」からそれぞれ継承関係を考えました。このときは「属性」と「操作(メソッド／操作)」のみに注目しましたが、今回はその他の特性(プロパティ)にも注目します。

多くの書籍やセミナーでは、継承関係を考える時にクラスの「属性」と「操作」のみに注目して継承を解説しています。これは間違いです。
また、「is-a」関係「a-kind-of」関係が成立することが、クラス間の継承関係の「正当性」の条件にしている書籍やセミナーもありますが、これも間違いです。

第3回のコラムでスーパークラスからサブクラスに継承される特性(プロパティ)を解説しましたが、クラス間の継承関係を検討するときは、継承される特性(プロパティ)全てについて考慮しなければなりません。

継承される特性(プロパティ)の全てを考慮する

プログラム言語毎に機能として直接サポートされている特性(プロパティ)は異なります。また自動的にスーパークラスからサブクラスに特性(プロパティ)を継承する機能も異なります。しかし、オブジェクト指向の「継承関係の理論」からは【表6-1】に掲載した特性(プロパティ)を検討することが原則になります。

継承される特性(プロパティ)	
● 属性名	● 操作名
● 属性の可視性(private／protected／public)	● 操作の可視性(private／protected／public)
● 属性のタイプ(型)	● 操作の種類(静的／非仮想／仮想／純粋仮想など)
● 属性の不変条件(制約)	● 引数の数とタイプ(型)と順番
● クラス不変条件	● 返値名とタイプ(型)
● 関連(関連端名)	● 操作の事前条件
● 関連の多重度と制約	● 操作の事後条件
● 関連(関連端名)の可視性(private／protected／public)	● 例外

【表6-1】

【表6-1】継承される特性(プロパティ)は、サブクラスで再定義により変更が可能なものもあります。再定義する場合は注意が必要です。

再定義の機能もプログラム言語毎のサポートによって多少異なりますが、特に多相(ポリモフィズム／多態)を利用する時に注意が必要になります。どのような事に注意する必要があるかについては今回のコラムで解説していきます。

継承関係にあるクラス間の「振る舞いの整合性」を考慮する

クラス間の継承関係を検討するときに重要となるのは、【表6-1】に掲載した特性(プロパティ)の「整合性」です。「継承の種類(コラム第2回と第4回参照)」は多数ありますが、どの種類の継承を利用する場合もスーパークラスからサブクラスに継承する全ての特性(プロパティ)を考慮します。そして、スーパークラスとサブクラスの「振る舞いの整合性」を正確にチェックする必要があります。そして「振る舞いの整合性」を判定します。

スーパークラスとサブクラスの**「振る舞いの整合性」**がある場合に**「継承関係の正当性(correctness)」**が保証されます。

これは「継承関係」の設計の核心です。
今回のコラムではこの核心部分の導入にあたる解説です。

スーパークラスとサブクラスの「振る舞いの整合性」

スーパークラスとサブクラスの「振る舞いの整合性」は、クラス間の継承関係では、継承される特性(プロパティ)の全てを考慮して「振る舞いの整合性」を判定しなければなりません。そのためには、まず継承される特性(プロパティ)を明確に意識することから始めなければなりません。

第5回のコラムで利用した長方形と正方形の例を用いて継承する特性(プロパティ)を説明することにします。第5回では特性(プロパティ)として属性と操作についてのみ解説の対象としましたが、今回はさらに操作の「事前条件(pre-condition)」と「事後条件(post-condition)」、および「クラス不変条件」の特性(プロパティ)を含めてについて検討します。これによりスーパークラスとサブクラスの「振る舞いの整合性」に基礎を理解することができます。

「モジュール」と「タイプ(型)」

第5回のコラムでクラスは「モジュール」と「タイプ(型)」の両方の性質を持つことを紹介しました。

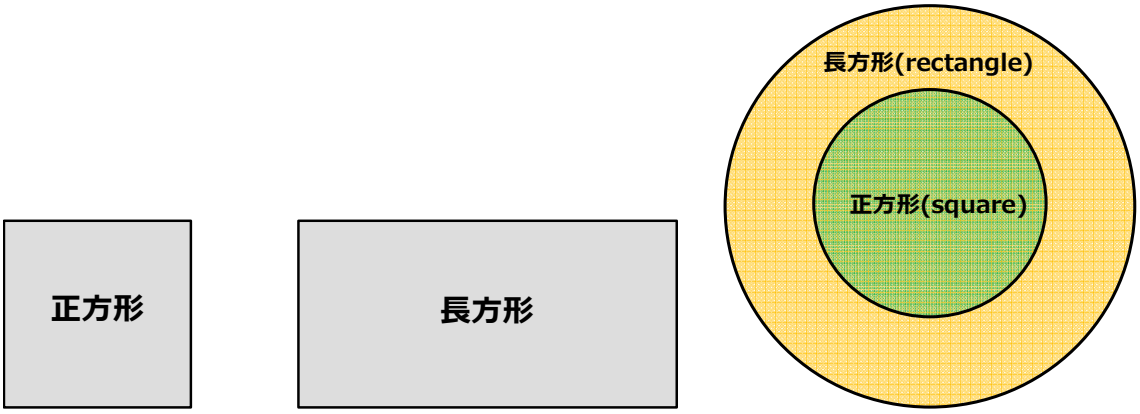
「モジュール」というのは、簡単に言ってしまえば「機能(処理)の集まり」です。一方「タイプ(型)」の方は文字通りタイプ(型)です。

クラスを「モジュール」としてみるか、「タイプ(型)」としてみるかの視点の違いで、継承を利用する時に鮮明になってくる違いがあります。

クラスが「データ抽象タイプ(型)」であり「モジュール」と「タイプ(型)」の両方の性質を持つ事は大きなメリットですが、一方で注意点も存在します。この注意点の１つがスーパークラスとサブクラスの「振る舞いの整合性」の判定になります。

「タイプ(型)」に焦点をおいた継承関係

まずは、クラスの「タイプ(型)」を焦点においた継承から見ていきます。「タイプ(型)」を焦点においた継承では、「長方形」と「正方形」において、どちらがより図形として一般的であるかを考えます。



【図6-1】

図形として「長方形」と「正方形」を考えると、【図6-1】のベン図のような関係を考えることができます。

「正方形」は「長方形」の一種であるからです。少し論理的な記述をすれば、

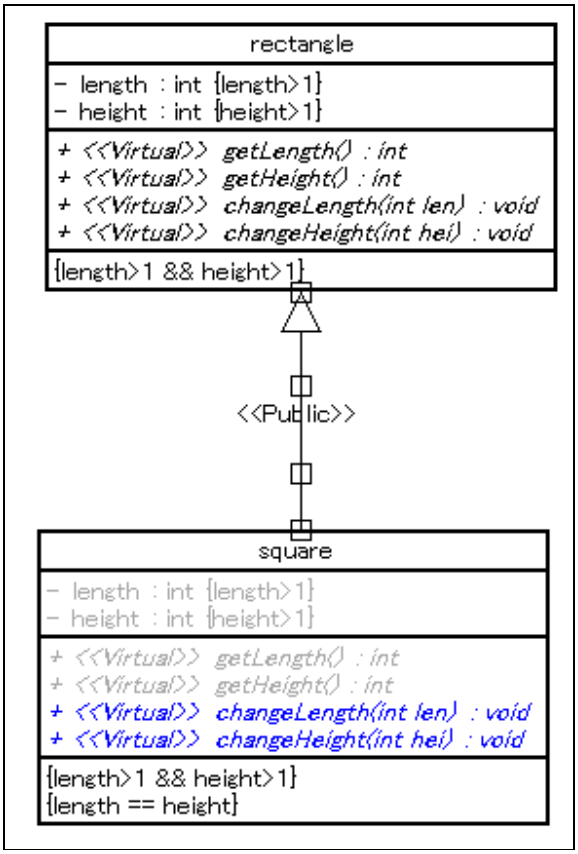
図形としての「長方形」と「正方形」の関係の形式的表記
● 「正方形」は「長方形」であるための十分条件(sufficient condition)である
● 「長方形」は「正方形」であるための必要条件(necessary condition)である

【表6-1】

と表現できます。

「属性の不変条件」と「クラス不変条件」

【図6-2】はクラス「rectangle(長方形)」とクラス「square(正方形)」を図形の種類(タイプ)に焦点を置いた継承関係です(【図6-3】も参照)。【図6-2】の継承関係は「is-a」関係あるいは「a-kind-of」関係があることは【図6-1】から自明です。

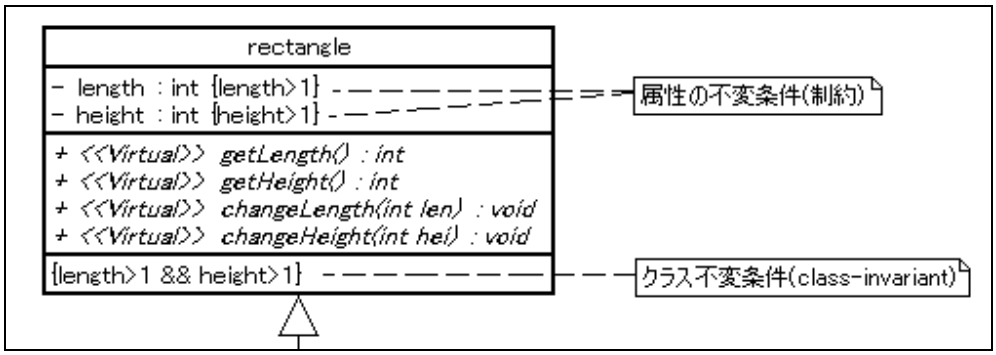


【図6-2】

【図6-2】ではクラス図の中にスーパークラスとサブクラスの「振る舞いの整合性」の判定に必要となる特性(プロパティ)を表示して示しています。UMLの標準の記法に従えば、サブクラスが継承した特性(プロパティ)は省略して表記しません。また、クラスの第四領域に「クラス不変条件(class-invariant)」を表示していますが、これもUMLの標準の記法を拡張して表示しています。

なお、コラムでは「振る舞いの整合性」の自動判定と表示処理のために「科学的モデリング専用ツール(特許出願済み／Patent Pending)」でモデリングをしています。

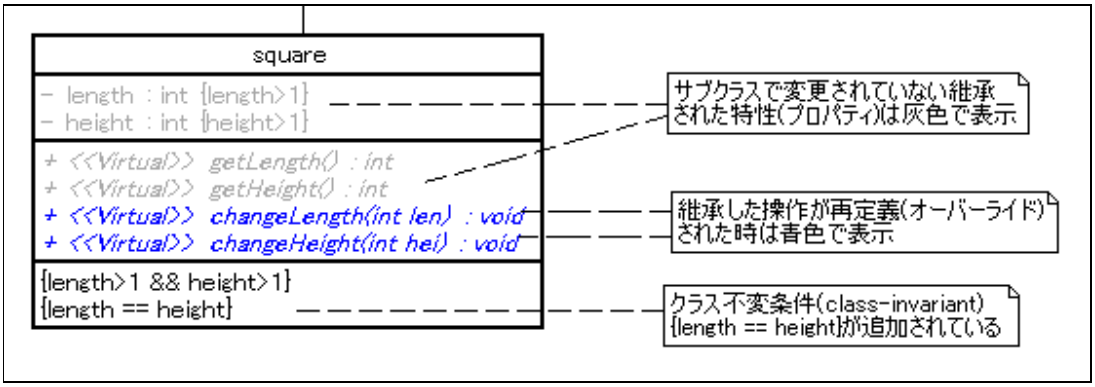
クラス「rectangle(長方形)」は、「横(Length)」「縦(Height)」のための２つの属性「length」と「height」を持っています。辺の長さが0ということはありませんから、属性「length」と「height」の不変条件として{length >1}および{height >1}が付与されています。



【図6-3】

クラス「rectangle(長方形)」には、属性を変更する操作として、「changeLength(int len):void」と「changeHeight(int len):void」が定義されています。この２つの操作は、「横(Length)」「縦(Height)」の辺の値を変更します。引数に渡らされた値で、クラス「rectangle(長方形)」の属性を更新する簡単な操作です。

次にクラス「square(正方形)」です。クラス「square(正方形)」はクラス「rectangle(長方形)」のサブクラスとして定義されています。



【図6-4】

まず違和感を覚えるのは、クラス「square(正方形)」の属性にクラス「rectangle(長方形)」から継承した属性「length」「height」があることです。正方形は辺の長さが等しい訳ですから、辺を表現する属性は本来１つであるべきです。

しかも属性名「length」「height」は、図形の正方形の特徴を表現する適切な属性名とは言えません。

正方形は辺の属性が本来１つで済むにも関わらず、クラス「square(正方形)」は属性名「length」「height」を持たざる負えないため、「クラス不変条件」として{length == height}を追加することになります。

「クラス不変条件」とは、クラスから生成される全てのオブジェクト(インスタンス)が満たさなければならない制約条件のことです(第3回のコラム参照)。このケースでは、クラス「square(正方形)」から生成される全てのsquare(正方形)オブジェクト(インスタンス)は、「横(Length)」「縦(Height)」の長さが等しくならなければならいという制約条件です。辺の長さのための属性が１つであれば、このクラス不変条件{length == height}を追加する必要はなかったはずです。

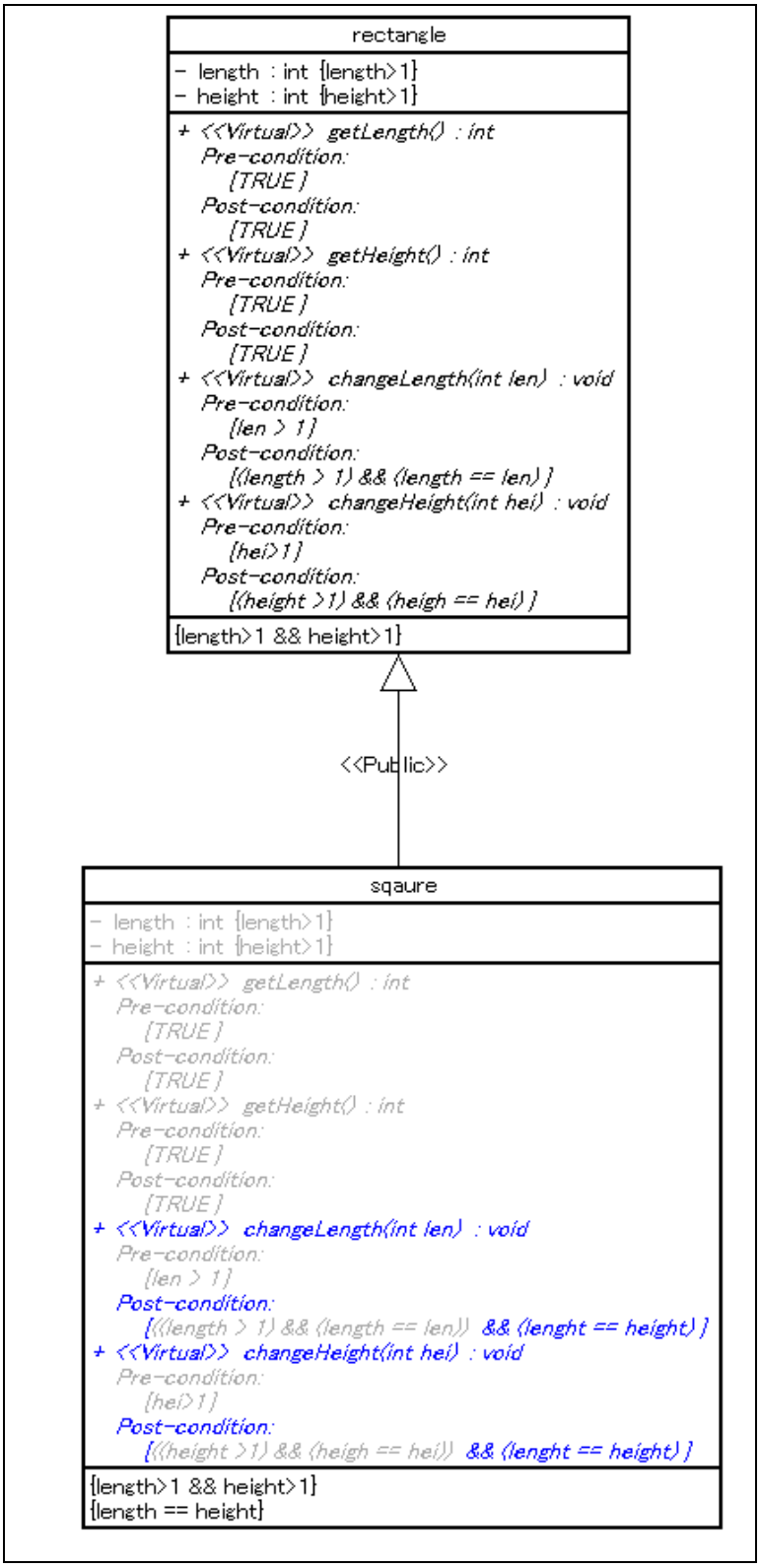
操作も考察してみましょう。操作「changeLength(int len):void」と操作「changeHeight(int len):void」はそれぞれ「横(Length)」「縦(Height)」の長さを変更する操作です。本来正方形は辺の長さが等しい訳ですから、辺の長さを変更する操作は１つで済みます。

しかし、クラス「square(正方形)」は、クラス「rectangle(長方形)」から操作「changeLength(int len):void」と操作「changeHeight(int len):void」を継承してしまいます。正方形の辺には「横(Length)」「縦(Height)」という概念がありません。また、属性のときと同様に操作名も正方形の特徴を的確に表現したものになっていません。

操作の「事前条件」と「事後条件」

ここまでは第5回のコラムの復習でしたが、ここからスーパークラスとサブクラスの「振る舞いの整合性」の判定を行うために、操作の特性(プロパティ)に注目していきます。

【図6-2】をより詳細なクラス図で表現すると【図6-5】になります。この図を見ながら検討していきます。

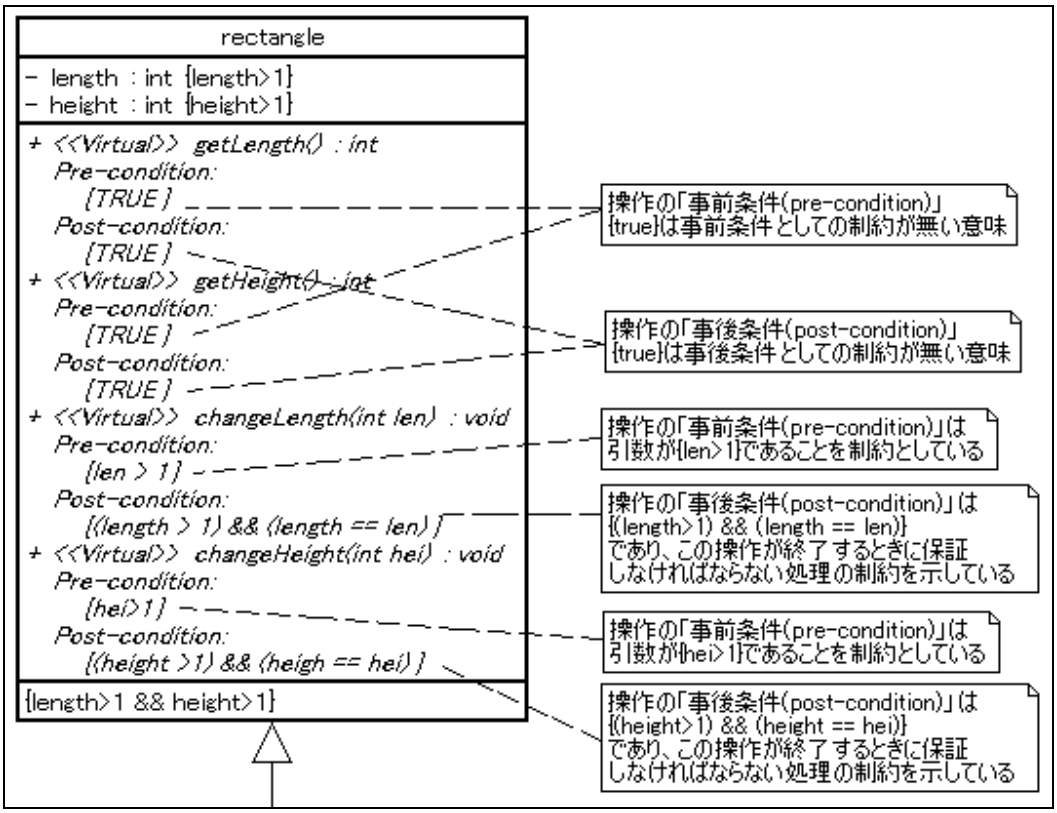


【図6-5】

【図6-5】では、操作の特性(プロパティ)として操作のシグネイチャ、可視性(public／protected／privateなど)、種類(静的／非仮想／仮想／純粋仮想など)および「事前条件(pre-condition)」「事後条件(post-condition)」を表示させています。「振る舞いの整合性」は、操作の可視性や種類も考慮して判定しますが、今回は「事前条件(pre-condition)」「事後条件(post-condition)」に注力して解説を進めていきます。

最初に理解すべきことは、「各操作には必ず「事前条件(pre-condition)」「事後条件(post-condition)」が存在する」とう事です。

【図6-6】をみてください。クラス「rectangle(長方形)」の「事前条件(pre-condition)」「事後条件(post-condition)」を確認します。



【図6-6】

操作「getLength():int」と操作「getHeight():int」は、それぞれ「横(Length)」「縦(Height)」の長さを返す参照操作ですが、「事前条件(pre-condition)」と「事後条件(post-condition)」が共に{true}と表示されています。これはこの２つの操作の「事前条件(pre-condition)」と「事後条件(post-condition)」において、制約の集合が∅であることを意味しています（簡単に言えば制約が無い訳ですが、数学的な言い方をすると、制約の集合が∅である「事前条件(pre-condition)」と「事後条件(post-condition)」が指定されていると表現します)。

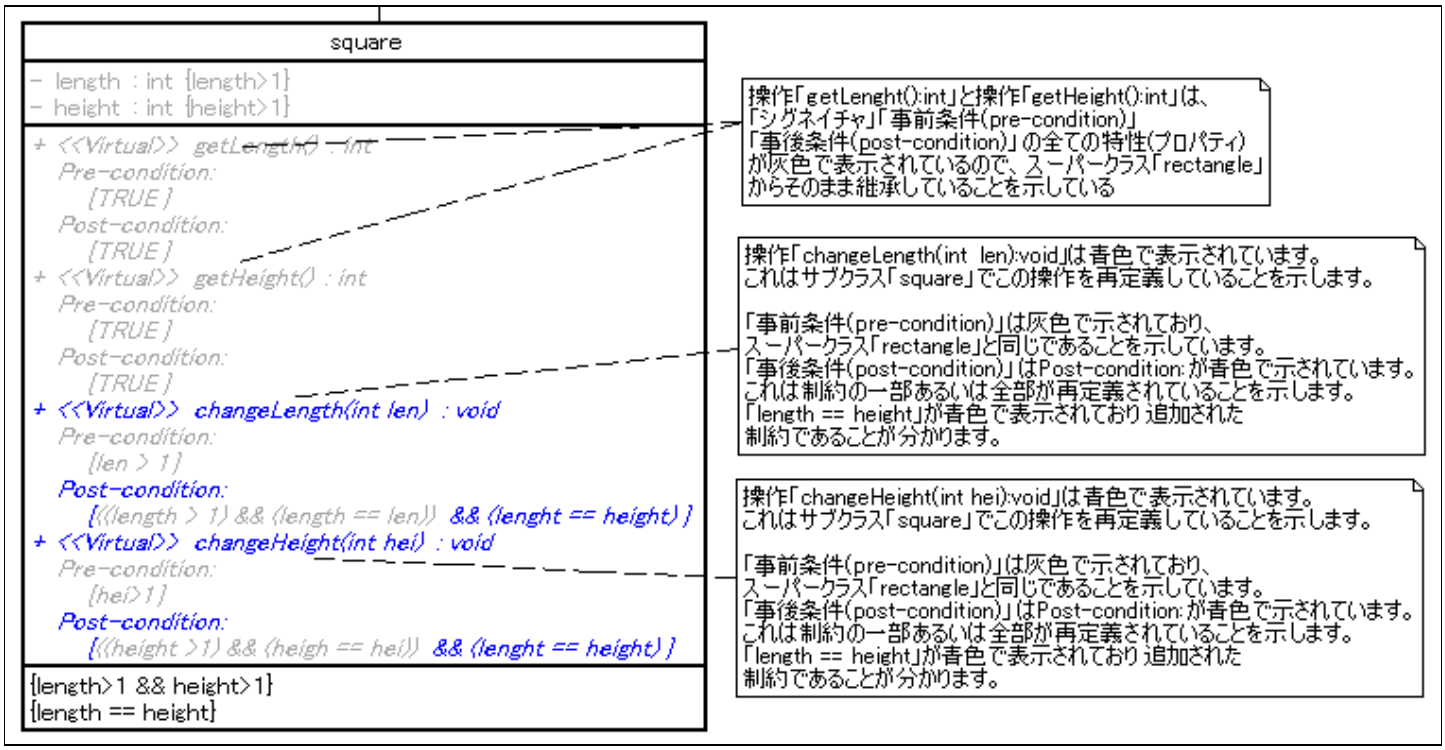
操作「changeLength(int len):void」の「事前条件(pre-condition)」「事後条件(post-condition)」も確認します。「事前条件(pre-condition)」に{len>1}が指定されています。この操作の引数の値が「>1」であることがこの操作を呼び出す条件です。これは、簡単に理解できると思います。「事後条件(post-condition)」は「(length > 1) && (length == len)」が指定されています。

つまり、{length > 1}と{length == len}の２つの制約が指定されており、２つの制約の両方を満たすことが指定されています。意味はこの操作の処理が終了するときに属性「横(Length)」の制約{len>1}を満たし、かつ、属性「横(Length)」の値が、引数で渡された値で更新されていることを操作が保証する必要がある制約を意味します。

操作の「事前条件(pre-condition)」「事後条件(post-condition)」は、操作の「意味的(振る舞い／処理)の仕様」です。操作のシグネイチャ(操作名、返り値と型、引数名と型と順番)は、「構文の仕様」です。

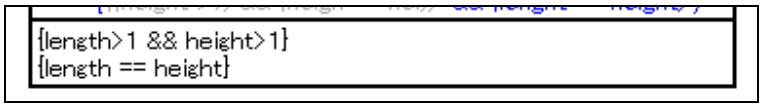
継承関係の「振る舞いの整合性」を判定するときは、操作の「構文の仕様」と「意味的(振る舞い／処理)の仕様」の両方が必要となります。

次は、サブクラス「square(正方形)」の操作の「意味的(振る舞い／処理)の仕様」である「事前条件(pre-condition)」「事後条件(post-condition)」を確認します。



【図6-7】

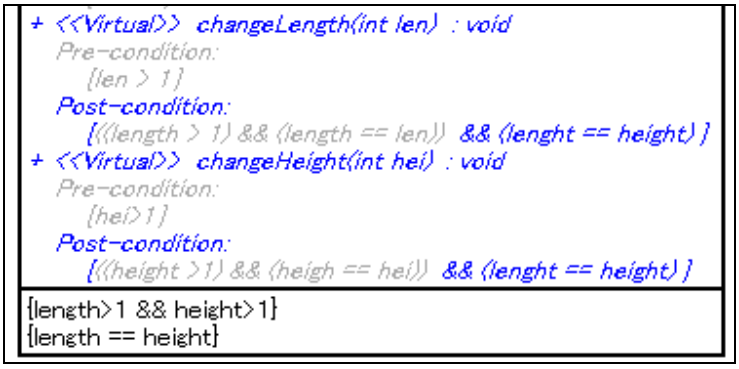
クラス「square(正方形)」は、全ての辺が同じ長さなので「クラス不変条件(class invariant)」に{length == height}を追加しています【図6-8】。この制約を満たすために継承した２つの操作のうち操作「changeLength(int len):void」と操作「changeHeight(int hei):void」を再定義(override)する必要があります。



【図6-8】

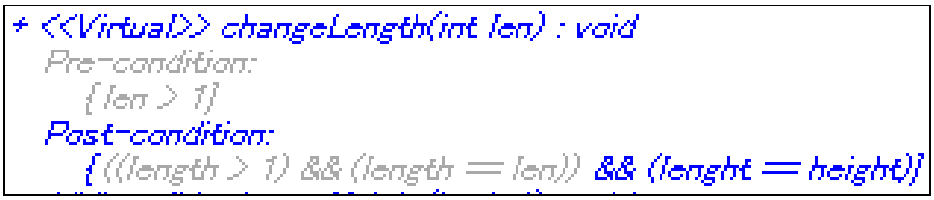
【図6-7】を見てみると、操作「changeLength(int len):void」と操作「changeHeight(int hei):void」が」は**青色**で表示されています。これはサブクラス「square」でこの操作を再定義していることを示します【図6-9】。

なお、今回使用するツールでは再定義している操作と再定義した条件式を自動判定する機能を使用してクラス図に**青色**で表示しています。



【図6-9】

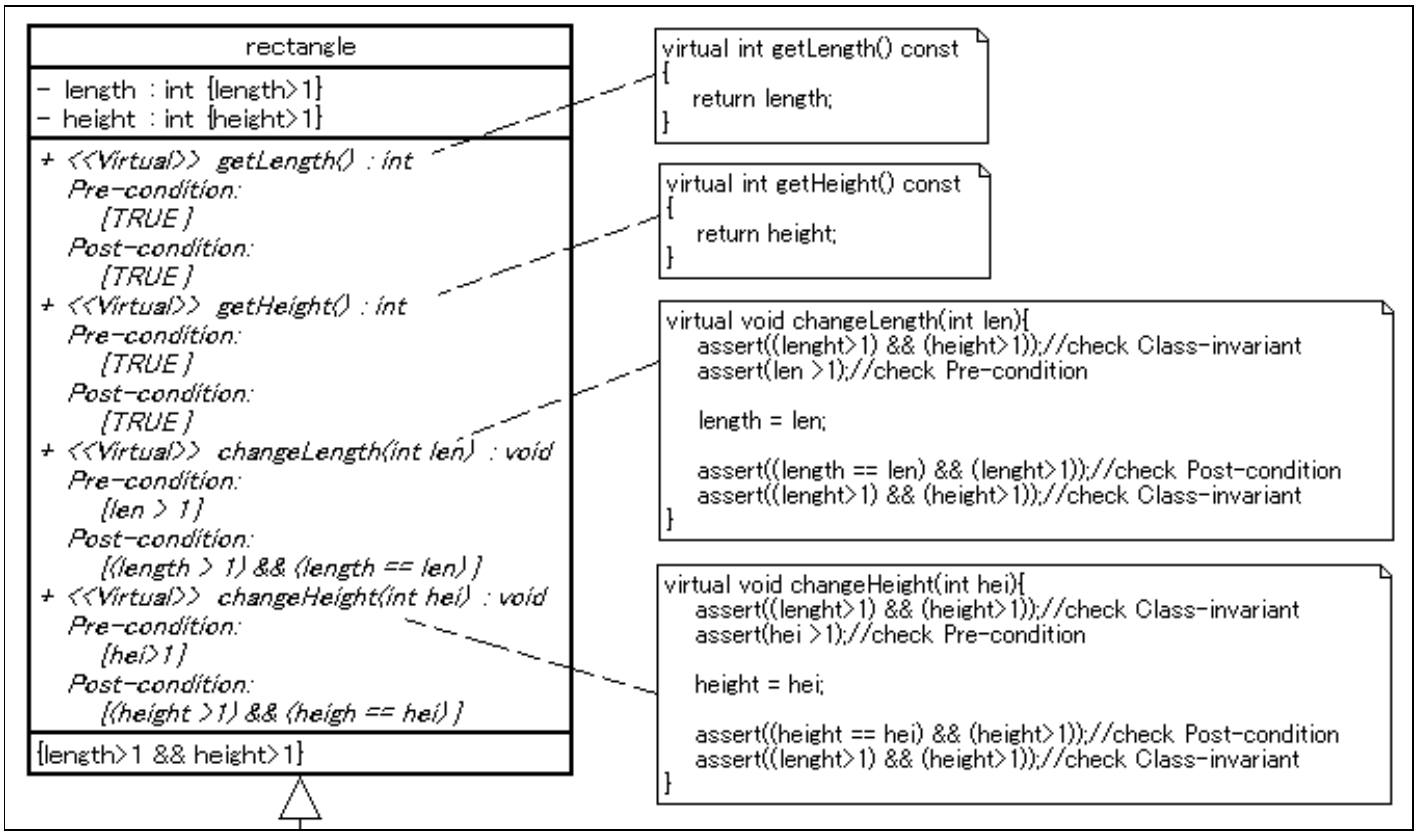
次にこの操作の「事前条件(pre-condition)」ですが、灰色で示されています。これはスーパークラス「rectangle」と同じであることを示しています。今回使用するツールでは再定義されずそのまま継承している属性、操作および条件式を自動判定する機能を使用してクラス図に**灰色**で表示しています。



【図6-10】

さらに、「事後条件(post-condition)」は「Post-condition:」の部分が青色で示されていますが、これは制約の一部あるいは全部が再定義されていることを示します。「length == height」が**青色**で表示されており追加された制約であることが分かります。

操作の処理のイメージをつかむために、【図6-11】と【図6-12】にコードイメージを併記したクラス図を示します。「科学的モデリング」のクラス図をどのように実装(プログラミング)するかは、今後のコラムで別途説明しますが、今回はコードイメージについて簡単に説明をしておきます。



【図6-11】

クラス「rectangle」のコードイメージで説明が必要と思われるのでは、各操作の「クラス不変条件(class-invariant)」「事前条件(pre-condition)」「事後条件(post-condition)」をコード化して挿入する部分です。

「クラス不変条件(class-invariant)」「事前条件(pre-condition)」「事後条件(post-condition)」は、実際に開発対象のプログラム言語でコード化されるのです。

「クラス不変条件(class-invariant)」「事前条件(pre-condition)」「事後条件(post-condition)」をコード化する方法にはいくつかのやり方があります。少し先になりますが本コラムで詳細に戦略的な実装方法を紹介する予定です。「クラス不変条件(class-invariant)」「事前条件(pre-condition)」「事後条件(post-condition)」には、テクニカルな実装方法が多数存在しますので詳細に解説したいと考えています。

ここでは「クラス不変条件(class-invariant)」「事前条件(pre-condition)」「事後条件(post-condition)」のコードイメージが分かる程度の単純かつ典型的な方法であるassertを使っています。

実開発においては、試験が終了後にコンパイルスイッチでassertを無効にすることで、「クラス不変条件(class-invariant)」「事前条件(pre-condition)」「事後条件(post-condition)」を取り除くことができますから、コードサイズや処理速度に影響を与えないで済みます。

「クラス不変条件(class-invariant)」「事前条件(pre-condition)」「事後条件(post-condition)」は、ある決まった規則で操作の中で記述します【表６-2】。また、コンストラクタと(C++の)デストラクタの中にも記述する必要があります。

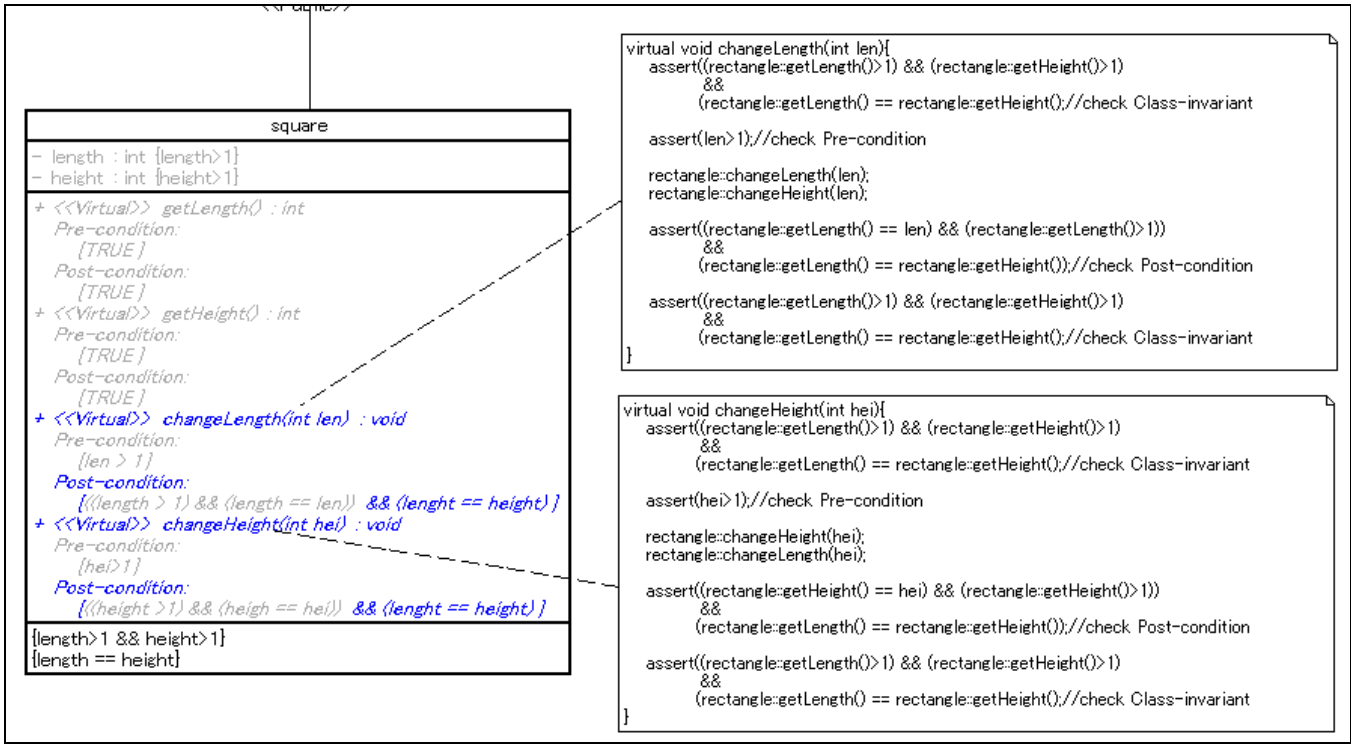
各操作における「クラス不変条件(class-invariant)」「事前条件(pre-condition)」「事後条件(post-condition)」のチェックコードの位置と順番

操作{
① 「クラス不変条件」チェックコード //操作開始直後に「クラス不変条件」が満たされているかチェック
② 「事前条件」チェックコード //次に「事前条件」をチェック
 	
<操作の処理コード>
 	
③ 「事後条件」チェックコード //操作処理後に「事後条件」をチェック
④ 「クラス不変条件」チェックコード //操作終了直前に「クラス不変条件」が満たされているかチェック
}

【表６-2】

コンストラクタとデストラクタのおいては、「クラス不変条件(class-invariant)」「事前条件(pre-condition)」「事後条件(post-condition)」のチェックコードの位置と順番が【表6-2】とは異なります。コンストラクタとデストラクタのチェックコードについては、本コラムで詳細に戦略的な実装方法を紹介するときに合わせて紹介いたします。

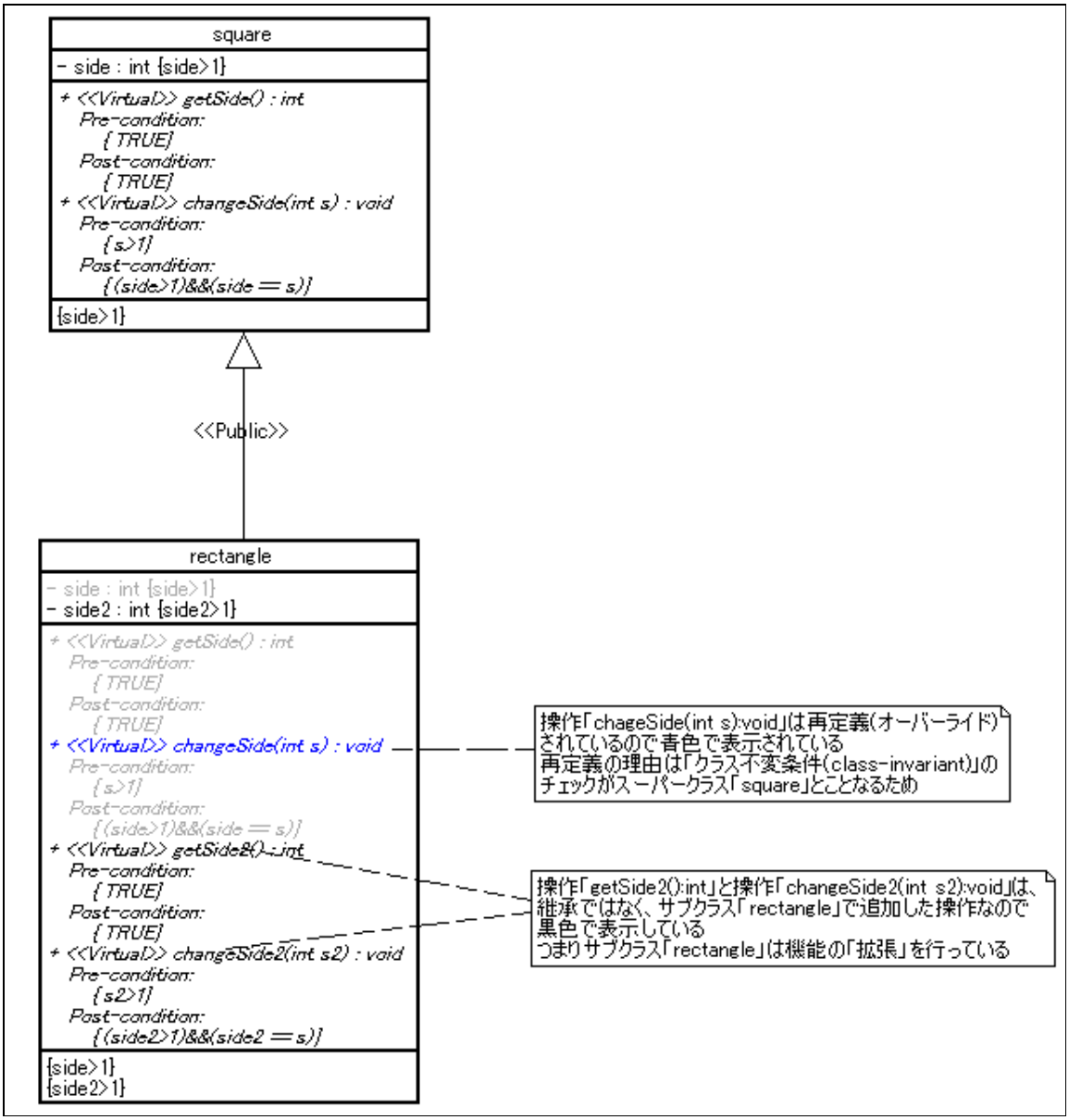
なお、「クラス不変条件(class-invariant)」「事前条件(pre-condition)」「事後条件(post-condition)」が守られなかったとき、は例外をスローすることが一般的ですが、例外の説明を今回は割愛します。例外は継承と密接な関連があるので、極めて重要で解説することが多数あります。そこで、例外と「クラス不変条件(class-invariant)」「事前条件(pre-condition)」「事後条件(post-condition)」の関係性について、これも後日コラムで詳しく解説します。



【図6-12】

モジュールに焦点をおいた継承関係

次はクラス「rectangle(長方形)」とクラス「square(正方形)」をモジュールの性質に重点を置いて継承関係を作成します。【図6-13】のようにクラス「rectangle(長方形)」がクラス「square(正方形)」から「属性と属性の不変条件」「操作」「クラス不変条件」を継承する関係になります。

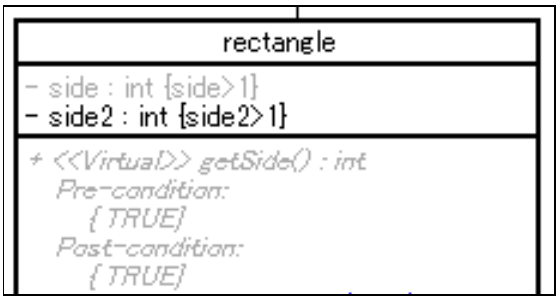


【図6-13】

オブジェクト指向では余分なコードを新規に作成でせず、再利用する大切な考え方があります。既にテスト済みのコードであれば再利用しない手はありません。このような視点にたてば【図6-13】のような継承関係も正当な継承関係の利用の１つです。

ただし、【図6-13】の継承関係はスーパークラスとサブクラス間に「is-a」や「a-kind-of」関係はありません。さらに【図6-2】のスーパークラスとサブクラスの継承関係とは、逆転した関係にあります。しかし、これが直接問題となることはありません。【図6-13】の図を見ながらより詳細に見ていきましょう。

サブクラス「rectangle(長方形)」が、スーパークラス「square(正方形)」からそのまま継承する特性(プロパティ)は灰色で表示しています。



【図6-14】

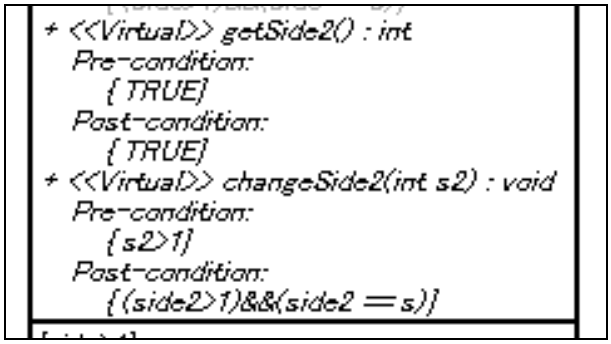
【図6-14】を見ると属性「side2:int{side2>1}」が追加されています。継承された属性でなく、サブクラス「rectangle(長方形)」が拡張により追加した属性であることが黒色で表示されていることで分かります。

参照操作「getSide():int」は、操作のシグネイチャおよび「事前条件(pre-condition)」「事後条件(post-condition)」の全てが灰色で示されています。これはサブクラス「rectangle(長方形)」がそのまま継承することを示しています。

【図6-15】

更新操作「changeSide(int s):void」は、操作のシグネイチャが青色に表示されています。これはこの操作が再定義(オーバーライド)されていることを示しています。再定義している理由と内容はこの後すぐにコードイメージを見ながら解説します。

この「事前条件(pre-condition)」「事後条件(post-condition)」は灰色で示されていますから、「事前条件(pre-condition)」「事後条件(post-condition)」はスーパークラスから継承した時の条件と同じです。結論として、この操作は「事前条件(pre-condition)」「事後条件(post-condition)」はスーパークラスと同じであり、操作の処理が再定義(override)されていることを意味します。



【図6-16】

残りの２つの操作「getSide2() : int」「changeSide2(int s2) :void」は、サブクラス「rectangle(長方形)」で追加した操作です【図6-16】。サブクラスで追加した特性(プロパティ)は黒色で表示しますから、２つの操作は継承した操作ではないことが分かります。

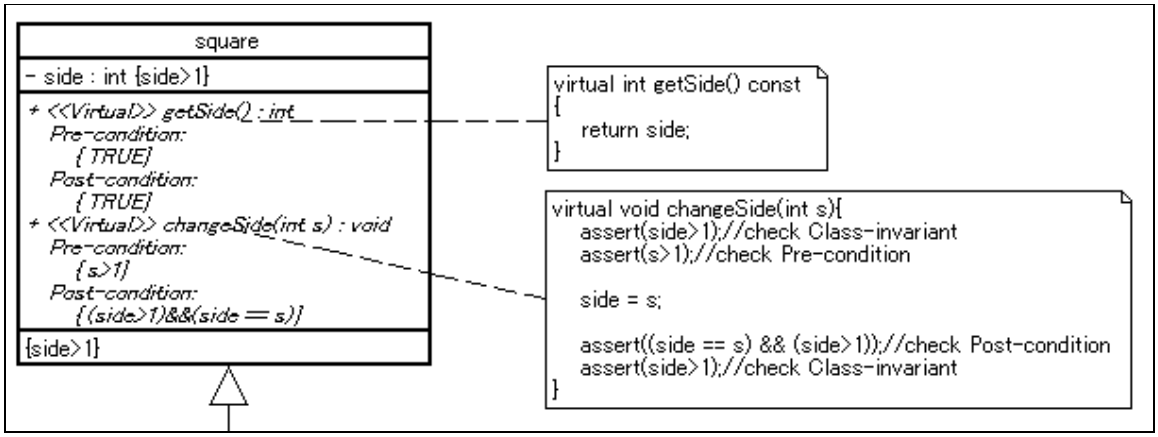
さて、ここで【図6-13】の継承関係を振り返り考察してみましょう。

クラス「square(正方形)」は辺の長さを表現する属性は１つで済みましたが、サブクラス「rectangle(長方形)」は縦と横の辺の属性が必要になりますから、属性「side2」を追加してクラスを「拡張(extension)」しています。ただし、長方形は辺として「横(Length)」「縦(Height)」をもつので、クラス「rectangle(長方形)」としては、本来であれば属性名「length」と「height」としたいところです。

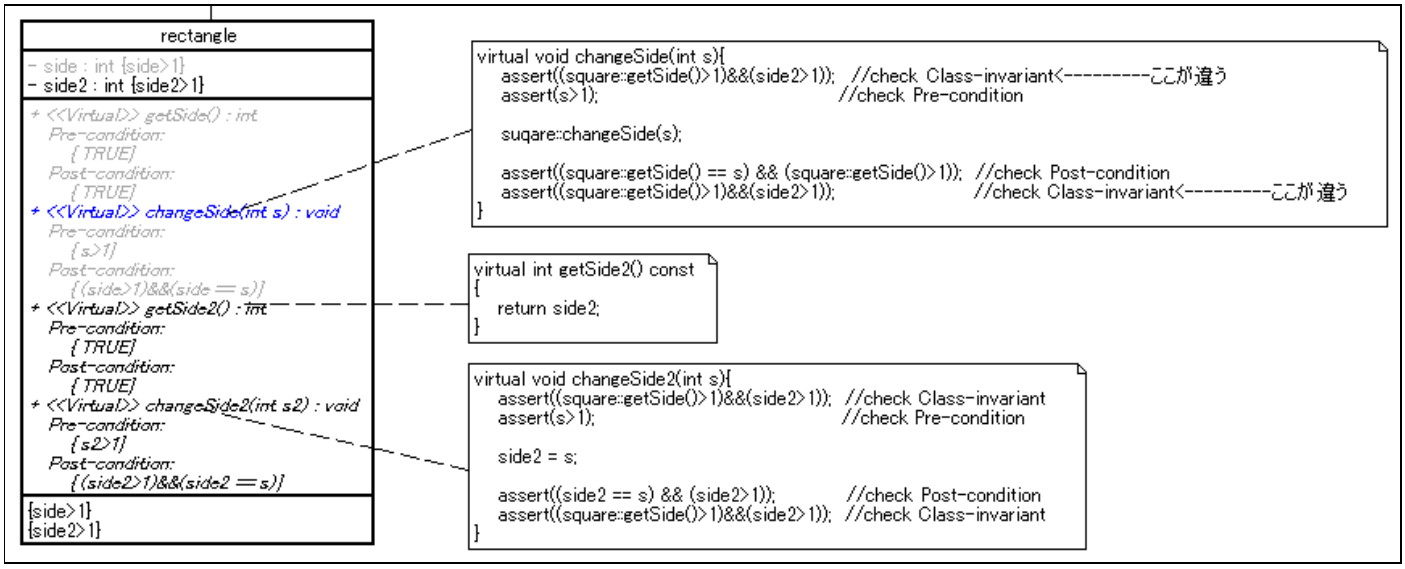
さらに、長方形は「横(Length)」「縦(Height)」の辺の長さが異なることために、クラス「rectangle(長方形)」は操作「getSide2() : int」と操作「changeSide2(int s2):void」追加して、ここでもクラスを「拡張(extension)」しています。

属性と全く同様に、長方形は「横(Length)」と「縦(Height)」の辺の長さを変更することをより的確に表現する操作名に本来はしたいところです。

【図6-17】【図6-18】が更新操作「changeSide(int s):void」のコードイメージです。



【図6-17】



【図6-18】

クラス「rectangle」のコードイメージのポイントを１つあげるとすると、更新操作「changeSide(int s):void」の箇所についてです。

クラス「rectangle」の「事前条件(pre-condition)」「事後条件(post-condition)」および操作の処理のコードは、スーパークラス「square(正方形)」と同じです。しかし、クラス「rectangle」の「クラス不変条件(class-invariant)」が

- {side>1}
- {side2>1}

なので、スーパークラス「square(正方形)」の「クラス不変条件(class-invariant)」とは異なっています。

そのため、assert文も下記の様になりスーパークラスとは異なったコードになります。

- assert((square::getSide()>1)&&(side2));

このassertコードをこの操作の最初と最後に挿入する必要があるので、操作「changeSide(int s):void」は再定義(オーバーライド／特殊化)される必要があります。そのためこの操作が青色で表示されている訳です。

「制約継承(restriction inheritance)」と「拡張継承(extension inheritance)」

言語設計者のBertrand Meyerは【図6-2】の継承を**「制約継承(restriction inheritance)」**、【図6-13】の継承を**「拡張継承(extension inheritance)」**と分類し名前をつけています([文献6-1]参照)。なお、Bertrand Meyerの分類は、継承種類の名称が彼独特の分類名を用いている点は注意してください。

【図6-2】「制約継承(restriction inheritance)」vs.【図6-13】「拡張継承(extension inheritance)」
● 「制約継承」とはサブクラスで**制約を追加する**継承関係： ➢ クラスをタイプ(型)として見れば、サブクラスで追加された制約を除けばスーパークラスとサブクラスタイプのタイプ(型)は**同型**です(第4回コラム参照) ➢ 逆に言えば、サブクラスで**制約を追加する継承**になります ➢ サブクラスは継承した操作を再定義(オーバーライド)しているので、スーパークラスのタイプ(型)をサブクラスで**「特殊化(specialization)」**していることになります(第4回コラム参照) ● 「拡張継承」とは特性(プロパティ)を追加し拡張を行う継承関係： ➢ クラスをタイプ(型)として見ればサブクラスで特性(プロアパティ)が追加されているのでサブクラスのタイプ(型)は**「拡張(extension)」**されています(第4回コラム参照) ➢ クラスをモジュールとして見れば属性と操作がサブクラスで追加されているので機能が増えて**「拡張」**がされています

【表6-3】

【図6-2】「制約継承(restriction inheritance)」、【図6-13】「拡張継承(extension inheritance)」を機能的な点から眺めてみると、それぞれ【図6－19】と【図6－20】になります。

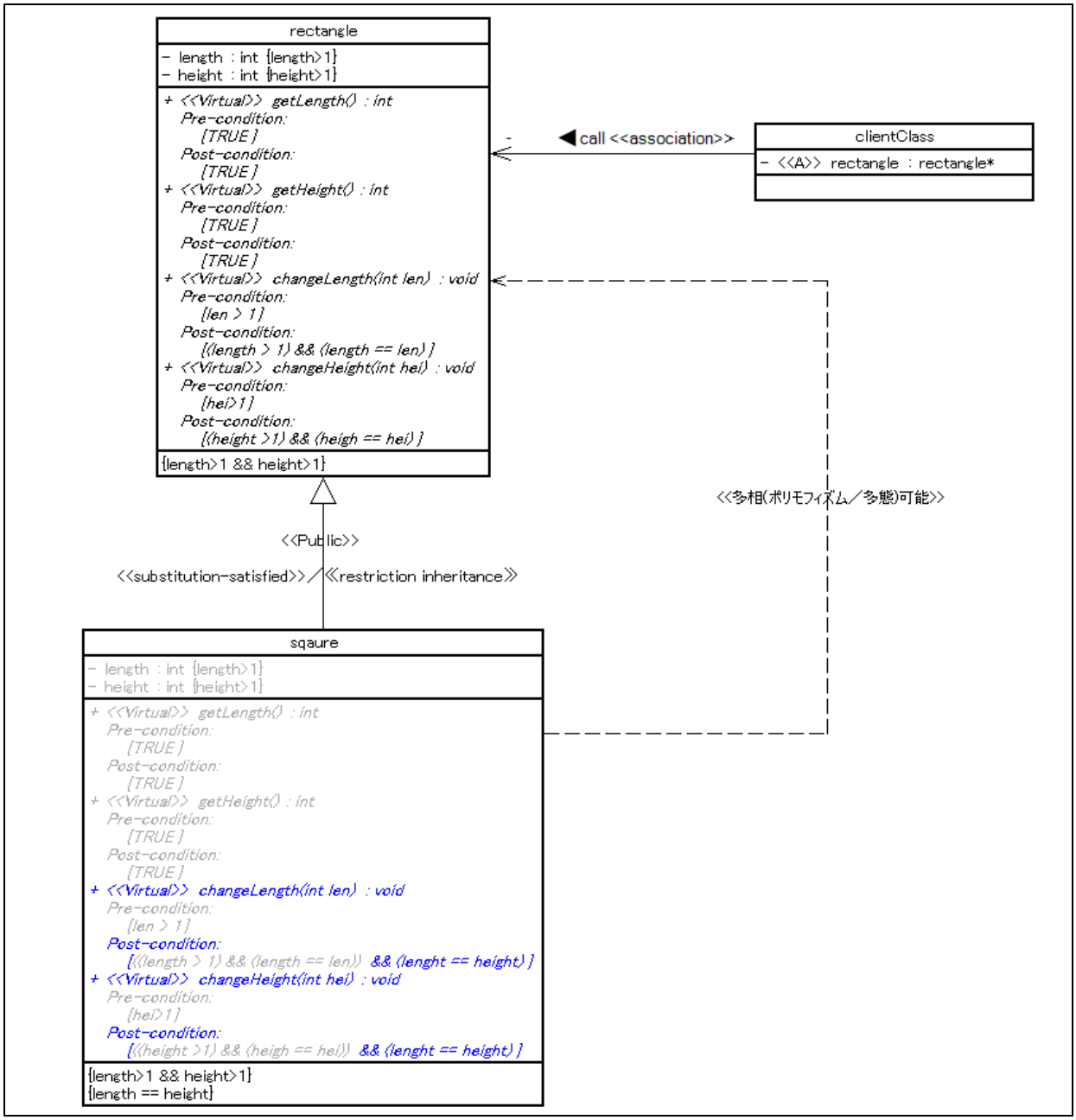
【図6－19】は【図6-2】の継承関係のスーパークラス「rectangle」に対してクラス「clientClass」が関連を持っており、クラス「rectangle」の操作を呼び出す事を示しています。

この継承関係はスーパークラス「rectangle」のオブジェクト(インタンス)にサブクラス「square」のオブジェクト(インタンス)を代入しても問題がありません。

スーパークラス「rectangle」とサブクラス「square」の「振る舞いの整合性」が完全に取れているからです。

既に見てきたように操作の「事前条件(pre-condition)」「事後条件(post-condition)」が一致しているからです。言い換えれば、全ての操作において「意味的な整合性がある」ので、多相(ポリモフィズム／多態)が問題無く成立します。

多相(ポリモフィズム／多態)を利用することを前提とした継承関係では、スーパークラスとサブクラスの操作の「振る舞いの整合性」が必要となります。

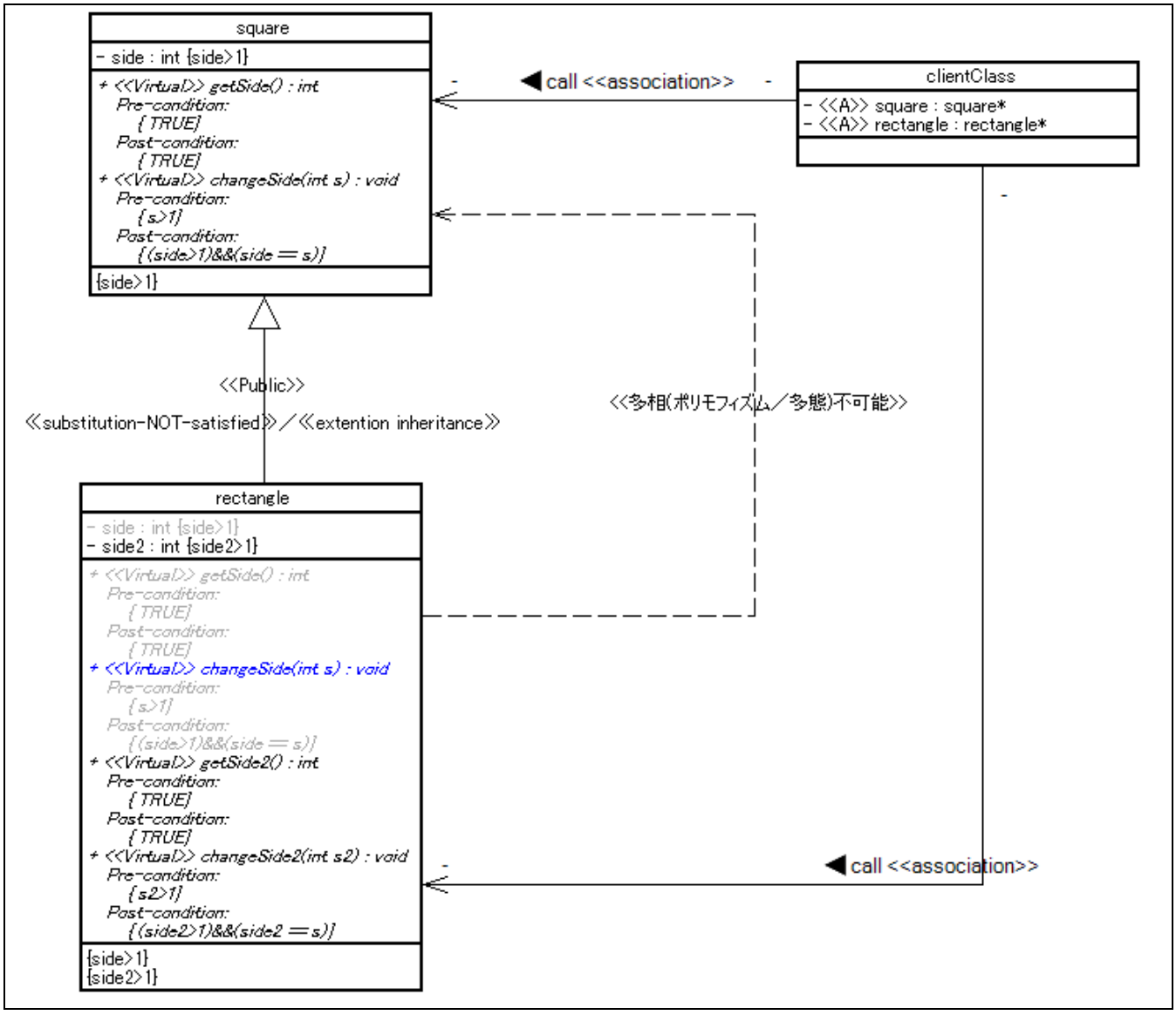


【図6-19】

一方、【図6－20】は【図6-13】の継承関係のスーパークラス「square」に対してクラス「clientClass」が関連を持っています。そして、クラス「square」の操作を呼び出す事を示しています。しかし、こちらの継承関係はスーパークラス「square」のオブジェクト(インタンス)にサブクラス「rectangle」のオブジェクト(インタンス)を代入が出来ないケースです。つまり、多相(ポリモフィズム／多態)が不成立となります。

それは、スーパークラス「square」とサブクラス「rectangle」の操作に「振る舞いの整合性」が無いからです。操作の「事前条件(pre-condition)」「事後条件(post-condition)」の条件式が不一致だからです。つまり操作の意味的な整合性が取れていません。

多相(ポリモフィズム／多態)を利用することを前提と“しない”、「構造」や「コードの利用」のための継承関係であれば、スーパークラスとサブクラスの操作の「振る舞いの整合性」が一致しなくても原則的に問題になりません。



【図6-20】

「継承関係の正当性」と「振る舞いタイプ(型)置換原理」

オブジェクト指向開発では「現実世界の構造をソフトウエアに取り込む」と言われますが、これを必要以上に単純に解釈してはいけません。私たちはソフトウエアを開発しているのであって、現実世界をモデルで模倣することが目的ではありません。最終的にシステムの処理を特定のプログラム言語で記述することと、「現実世界の構造」には大きな隔たりがあります。また、「現実世界の構造」といってもあくまで人間が属人的な視点を通じて構造を作成したものに過ぎません。

多くの書籍やセミナーで「is-a」「a-kind-of」関係を基準に継承関係を決定することを説明していますが、これは継承関係の設計や実装に感覚的で曖昧な方法を持ち込む稚拙な開発アプローチになってしまいます。

オブジェクト指向の専門家・実務家のRobert C. Martinは、彼の書籍([文献6-2])の中で、「is-a」「a-kind-of」関係を基準に継承関係を決定することを：

- 「極めて短絡的な思考プロセスであり、重大な問題を引き起こす」

と強く戒めています。

クラス間の継承関係の「正当性(correctness)」は、スーパークラスとサブクラス間に「is-a」関係や「a-kind-of」関係が存在の有無で決まる訳ではありません。

継承関係を利用する「目的」と「得たい効果」を明確にし、その上でどの「継承関係の種類」を決定によって決まります。

特にスーパークラスのオブジェクト(インスタンス)に、サブクラスのオブジェクト(インスタンス)の代入をする多相(ポリモフィズム／多態)を用いる場合は、クラスの「振る舞いの整合性」に意識を向けることが大切です。そして、このような考え方を**「振る舞いタイプ(型)置換原理(behavior subtype substitution principle)」**と呼びます。

「振る舞いタイプ(型)置換原理」を適用するには、クラスの「クラス不変条件(class-invariant)」と各操作の「事前条件(pre-condition)」「事後条件(post-condition)」を明確にすることがまず必要です

そして、継承関係にあるクラス間の「クラス不変条件(class-invariant)」「事前条件(pre-condition)」「事後条件(post-condition)」を比較して「整合性」を判断することが必要となります。

継承関係の正当性は開発者の「 (継承関係を使う) 立場」と「目的」に依存し決まります。この「 立場」と「目的」を明確にせず継承関係の「正当性(correctness)」を議論することは不毛です（[文献6-2] 参照）。

まとめ＆次回

今回は「クラス不変条件」「事前条件」「事後条件」を使った継承関係の正当性を判断する方法を紹介しました。

そして、このことをクラスの操作の「振る舞いの整合性」を通して解説しました。

最後に**「振る舞いタイプ(型)置換原理(behavior subtype substitution principle)」**という原理を紹介しましたが、これは、今回解説した「振る舞いの整合性」を発展させたものです。

「振る舞いタイプ(型)置換原理」について、具体的な解説を次回以降から進めていきます。

「振る舞いタイプ(型)置換原理」を理解すると、「継承を深く理解」し「効果的に使いこなせる」ことが可能になります。継承の効果を劇的に発揮できる技法がマスターできるからです。そして何より作成するモデルの完成度が高まり、品質や生産性が劇的に向上します。品質や生産性が劇的に向上する理由は、ソースコード開発が容易くなり、試験も非常に楽になるからです。その具体的な理由と詳細な技法解説は、今後のコラムの中で解説していきます。

参考文献

- 文献[6-1] [Bertrand Meyer 1997] Object-Oriented Software Construction(邦訳「オブジェクト指向入門 第2版-原則・コンセプト」「オブジェクト指向入門 第2版-方法論・実践」)
- 文献[6-2] [Robert C.Martin 2002]Agile Software Development, Principles, Patterns, and Practices (邦訳「アジャイルソフトウェア開発の奥義 第2版 オブジェクト指向開発の神髄と匠の技」)
- 文献[6-3] [Ian Joyner 1999] Objects Unencapsulated: Java、 Eiffel and C++??(邦訳「オブジェクト指向言語のはなし」)
- 文献[6-4] [Marshall P. Cline,Greg Lomow,Mike Girou1998]C++ FAQs (2nd Edition) (邦訳「C++FAQ C++プログラミングをきわめるためのQ&A集」)
- 文献[6-5] [Kayshav Dattatri 1997] C++: Effective Object-Oriented Software Construction: Concepts, Practices, Industrial Strategies and Practices(邦訳「C++実践オブジェクト指向プログラミング」)