UL Systems, Inc.

アナパタ勉強会
第10章
デリバティブ（金融派生商品）
10.4～10.5

2002年4月12日

矢崎博英@ウルシステムズ株式会社

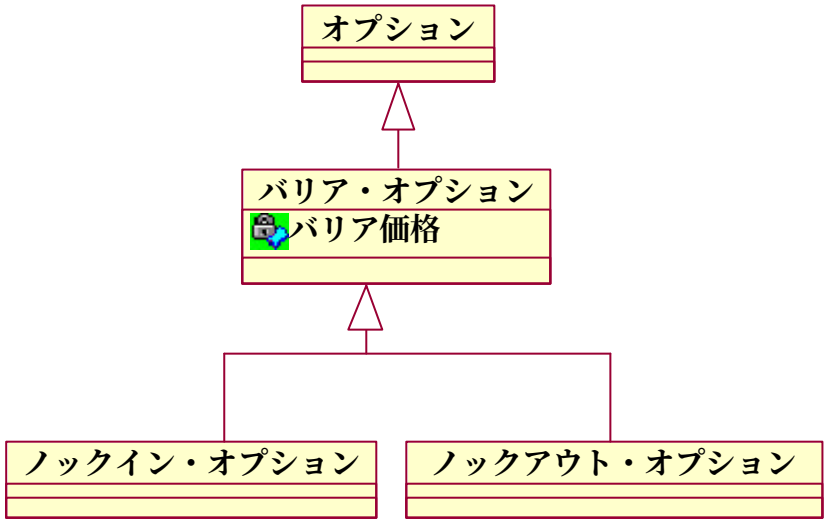
バリア・オプション

- 原資産の価格が満期日までに、ある設定された価格（バリア価格）に到達したかどうかで、オプションが発生したり消滅したりするオプション。大きくはノックイン・オプションとノックアウト・オプションに分かれる

- ノックイン・オプション
  - あるバリア価格を設定し、原資産価格が満期までの間にその価格に達しない場合にはオプションが発生しない、つまりオプションがその効力を発揮できない」という条件のついたオプション。一度達すると、その後価格がどのように変化しようともオプションは消滅しない

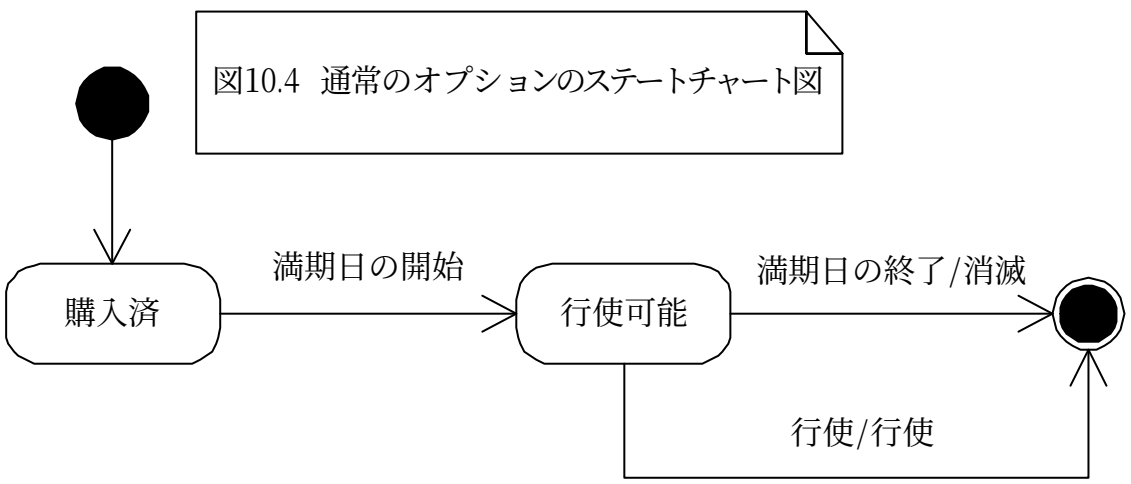
- ノックアウト・オプション
  - 原資産価格が満期までの間にバリア価格に達しなければオプションは有効だが、バリア価格に達するとオプションが無効となってしまう」という条件のついてオプション

バリア・オプションの概念モデル

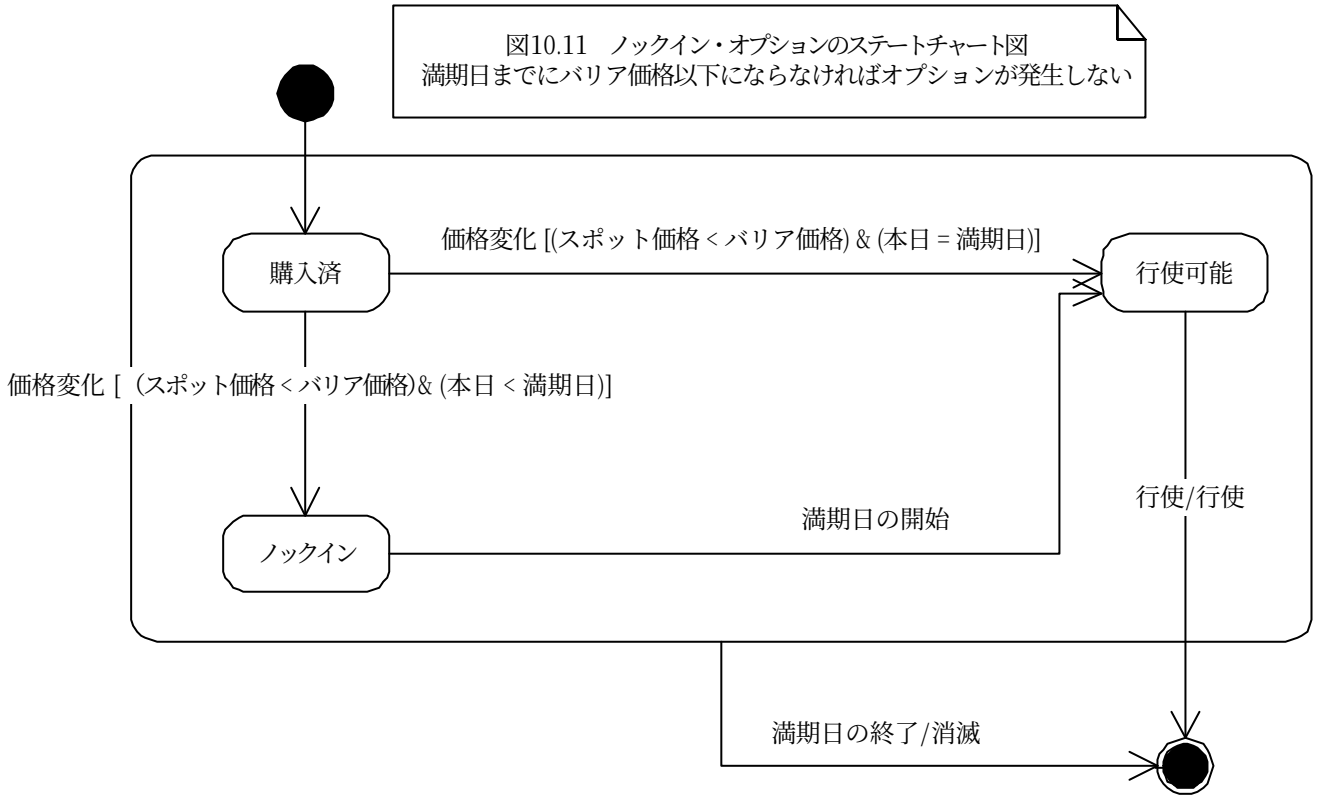


- バリア・オプションに属性「バリア価格」を追加する

通常のオプションのステートチャート



ノックインオプションのステートチャート図



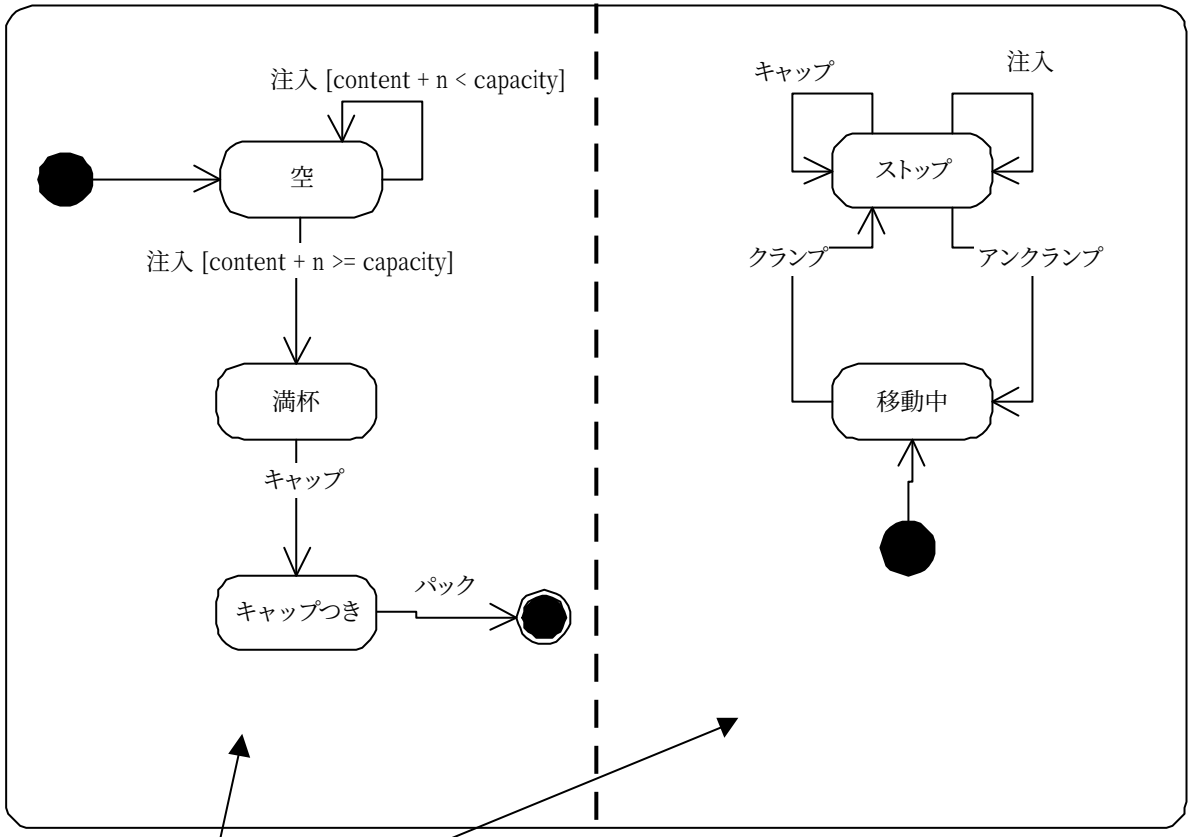
- サブタイプのステートチャートとして、この図は許されるか？

スーパタイプとサブタイプ

- ほとんどの（オブジェクト指向）方法論では、サブタイプがスーパタイプの代用になり得ることの重要性を強調している
  - ⇒ 「契約による設計」後ほど検討）
- 代用性 → オブジェクト（クラス）の場合、関連を追加することだけを許している（除去は許していない）点に、このことが反映されている
- ステートタイプにおいて代用性とは？
  - ShlaerとMellor
    - スーパタイプかサブタイプか**どちらか**にしかステートチャート図を描いてはいけない
  - Rumbaugh
    - サブタイプにはスーパタイプと直交する（orthogonal）ステートチャート図しか描いてはいけない

参考 :直交するステートチャート図

ボトルのステートチャート図



- お互い独立して変化する状態の遷移

クックとダニエルズによる議論

- サブタイプのオブジェクトは
  - スーパタイプのステートチャート図を保有していると考える。すなわち
    - スーパタイプのステートチャート図で定義されているステートになり得る
    - スーパタイプのステートチャート図で定義されているイベントに対して、そのイベントに反応するステートにある場合は、必ず反応しなければならない
    - 同じステート、同じイベントに対しては、トランジションが同じステートか、そのサブステートに向かわなければならない

- 上記に矛盾しない限りにおいて、新しい状態遷移を定義できる

※ 「Design Object Systems—Object Oriented Modeling With Syntropy(By Steve Cook & John Daniels/Printice Hall 1994」より

ファウラーの解釈

- クックとダニエルズによる議論は、すなわち 「サブタイプのステートチャートは2通りの方法で拡張できる」
  - サブタイプに直交するステートチャート図を置く
  - スーパタイプのステートにサブステートを定義し、スーパタイプのステートチャートのトランジションをそのサブタイプに直接向かうようにリダイレクトする

これらにプラスして、スーパタイプのステートに関係のない状態遷移をかけるというのがCook＆Danielsの意見のような気がするが、、、

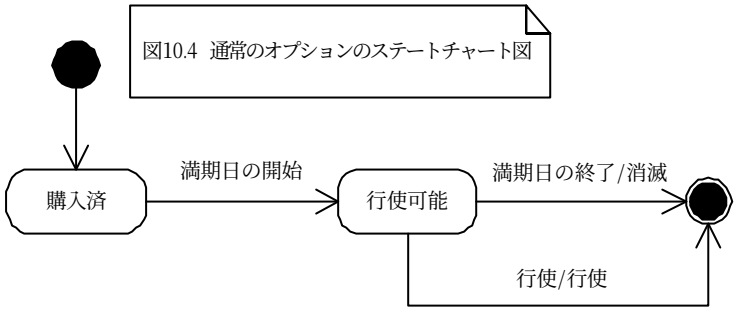
- クックとダニエルズの議論は 契約による設計」原理に沿ったものである

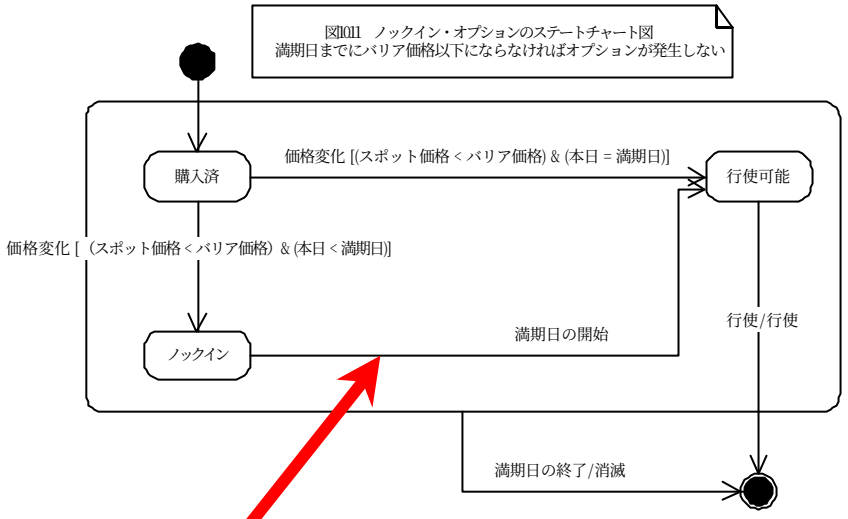
参考：契約による設計」の原理

- サブタイプのプレコンディションは、スーパタイプのそれより強くすることはできない。同じか、または弱くすることしかできない

- サブタイプのポストコンディションは、スーパタイプのそれより弱くすることはできない。同じか、または強くすることしかできない

図10．11の問題点（DBCの観点より）





満期日の開始イベントのソースステートが「購入済」でない

もう1つ「満期日の終了/消滅」をファウラー氏は問題だとしているが、そうではないと思う

図10．11の問題点（DBCの観点より）

- 満期日の開始イベントのソースステートが「購入済」でない

■オブジェクトがあるイベントに対して何もできない状態にあるときに、そのイベントを受け取ったら、オブジェクトは何をすべきか。。を考えることから生まれる疑問

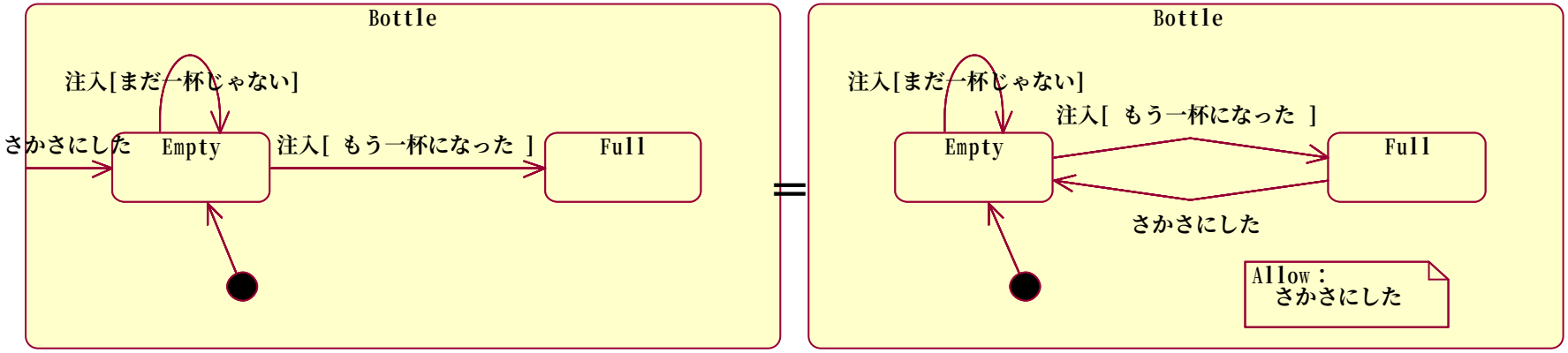
ポイントがずれてない？
（ウォウラーの議論が混乱しているように見受けられる）

実際には、以下の要件がみたされていないということでは

スーパタイプのステートチャート図で定義されているイベントに対して、そのイベントに反応するステートにある場合は、必ず反応しなければならない

参考 許されるイベント

- Allowed events (By Cook & Daniels)



- 「さかさにした」イベントをAllowed eventにするとは、そのイベントはどのステートにおいても受け入れるということである
- そのイベントに対して明示的なトランジションがない場合、セルフトランジションが起きると見なされる

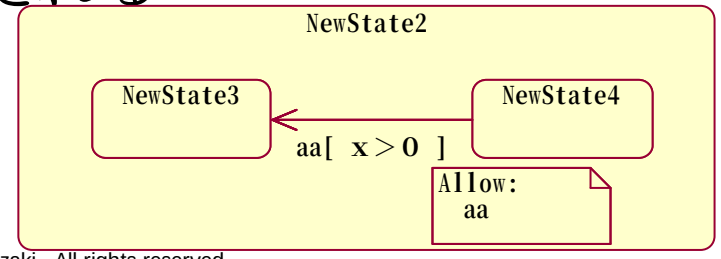
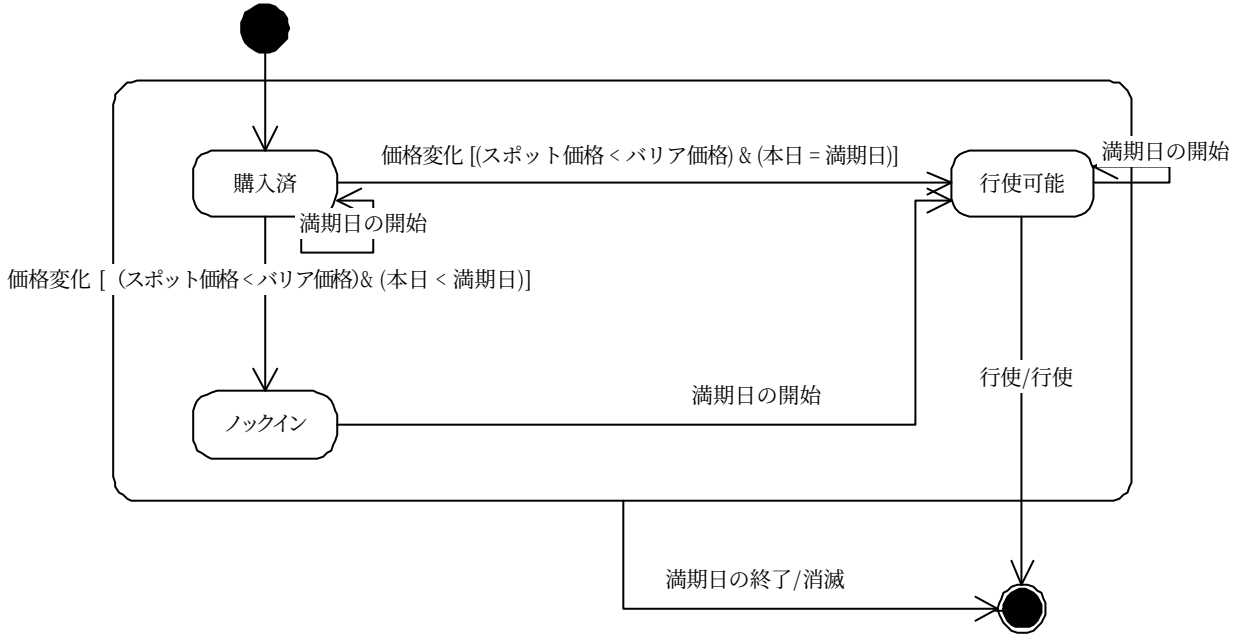


図10．11にAllowed　eventsを追加するとは、、

こいつと同じこと



これで以下の要件は満たされた

スーパタイプのステートチャート図で定義されているイベントに対して、そのイベントに反応するステートにある場合は、必ず反応しなければならない

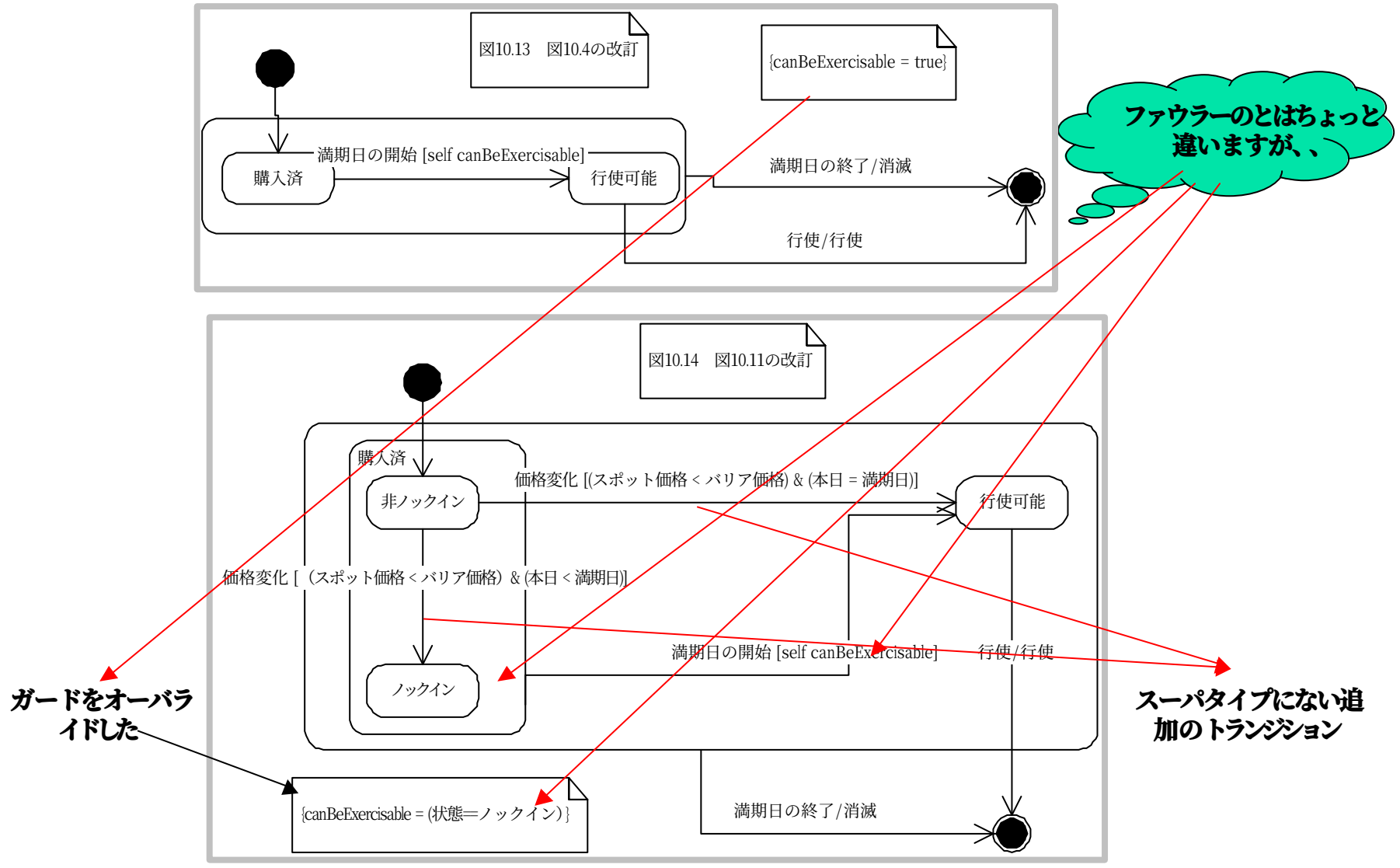
図10．11の問題点（DBCの観点より）

- 満期日の開始イベントのソースステートが「購入済」でない

■前の図でも問題がまだ残る。それは「同じステート、同じイベントに対しては、トランジションが同じステートか、そのサブステートに向かわなければならない」という要件を満たしていないという点

設計による契約によれば、ポストコンディションは弱めることはできない。ここでは、「購入済」状態にあって「満期日の開始」イベントを受け取ったとき、スーパタイプでは「行使可能」状態に遷移するのに、サブタイプで同じ状態に遷移しないのは、ポストコンディションを弱めていることではないか、という主張である

だったらこうしよう 図10. 13と14)



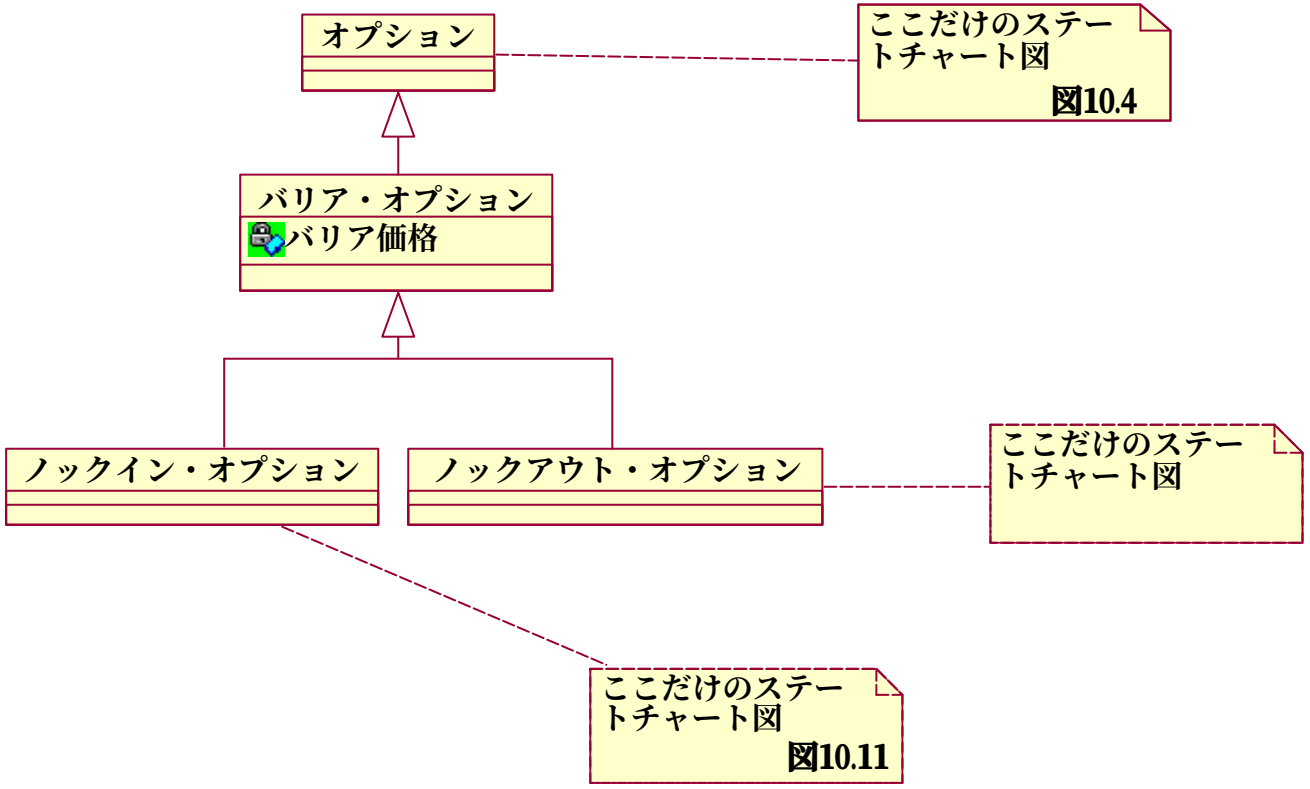
適合性（DBC的な）を使用する際の問題

- 図10．4や図10．11のほうが図10．13や図10．14より、振る舞いの記述が単純かつ明快である

- サブタイプ（ここではノックイン）を追加すると、（適合性を考慮して）スーパタイプのステートチャート図を変更した
  - 新しいサブタイプを追加するたびに起こりえる

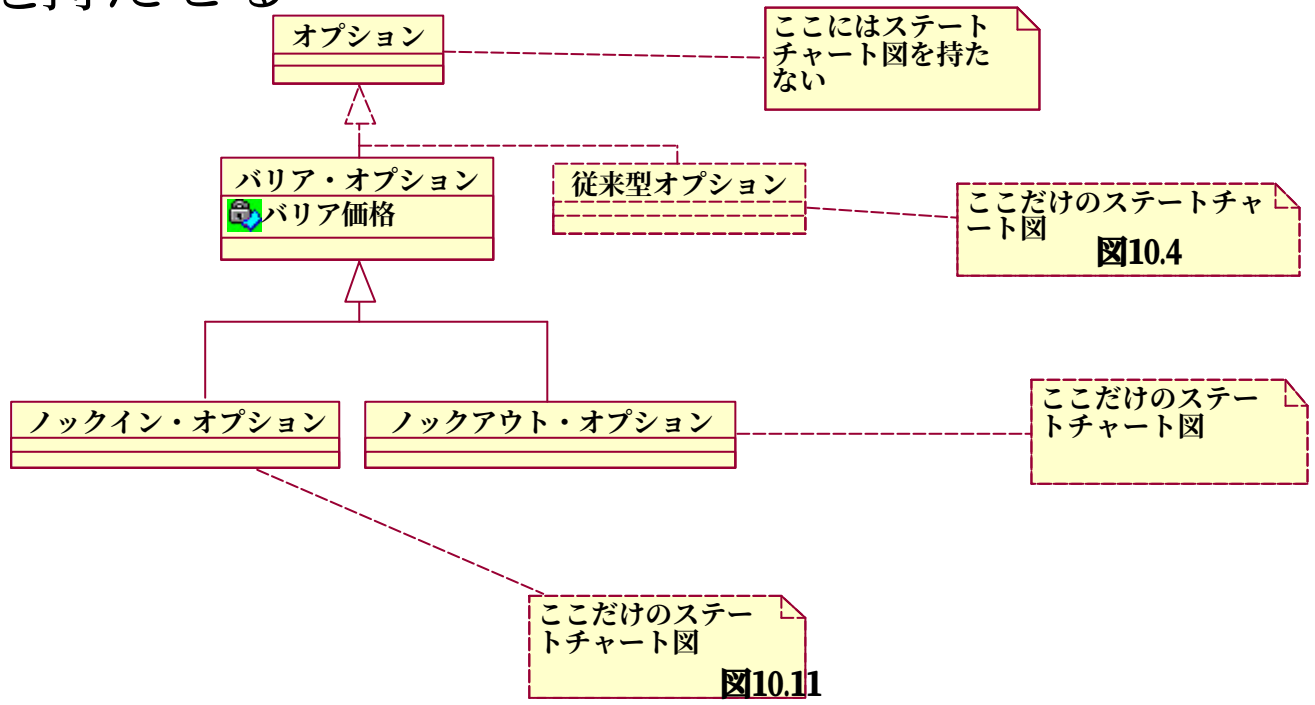
適合性（DBC的な）を使用する際の問題

- 解決案その１‥もう、適合性をとるのはやめよう！そんな細かいこと気にしない、気にしない‥



適合性（DBC的な）を使用する際の問題

- 解決案その２・・オプションを抽象型とし、そこにはステートチャート図を持たない。「従来型オプション」という型をサブタイプとして作る。「従来型オプション」「ノックイン」に別々のステートチャート図を持たせる



適合性（DBC的な）を使用する際の問題

- 解決案 1あるいは 2により一応解決できる。しかし、さらに深く考察してみよう

- 「契約による設計」では、サブタイプはそのスーパタイプのポストコンディションを満たさなければならないと主張している（サブタイプはスーパタイプのポストコンディションを弱めてはならない）
  - サブタイプはスーパタイプを代理できなければならない

- しかし、そのことは、ステートチャート図において同じイベントにおいてその「状態の遷移」先も同じでなければならない、という意味だろうか
  - Yes
  - No
  - あなたはどちら？？

適合性（DBC的な）を使用する際の問題

- ファウラー氏はNoという立場
- Design by contract says that subtypes must satisfy their supertype's postconditions. However, that does not necessarily imply that the postcondition on start of expiration date should include the transition to the exercisable state.

※アナパタ原著P.215

- サブタイプのオブジェクトは
  - ~~スーパタイプのステートチャート図を保有していると考える。すなわち~~
    - ~~スーパタイプのステートチャート図で定義されているステートになり得る~~
    - スーパタイプのステートチャート図で定義されているイベントに対して、~~そのイベントに反応するステートにある場合は、~~必ず反応しなければならない
    - ~~同じステート、同じイベントに対しては、トランジションが同じステートか、そのサブステートに向かわなければならない~~

適合性（DBC的な）を使用する際の問題

- ポストコンディションに対する2つの異なる考え方
  - 考えその１
    - ポストコンディションに、観察可能な状態への全ての変化を定義すべきである
    - この考えだと、ポストコンディションに定義されない状態変化はない
    - ゆえに、ステートチャート図のトランジションは全てポストコンディションに定義されたものとなる
  - 考えその２
    - ポストコンディションには、操作の終了時に必ずそうなっていなければならない状態を定義する。
    - この考えだと、ポストコンディションに定義されない状態変化を許す
    - ゆえに、ステートチャート図のトランジションは、ポストコンディションに定義されたものもあるし、そうでないものもある
      - ステートチャート図はポストコンディションを表すものではない

適合性（DBC的な）を使用する際の問題

- ファウラーの意見
  - 考え方その１（制限の強い考え）は、初めから全てのサブタイプを予見できないという点から適用が難しい
    - サブタイプが増えるたびにスーパタイプのステートチャート図を変更しなければならない
    - あるいは、スーパタイプのステートチャート図を変更できないので、サブタイプの柔軟性を犠牲にしなければならない
    - あるいは、初めから少し不自然なモデルにしなければならない（図10．16）。。。
  - よって、考え方2のほうがよい
    - スーパタイプのポストコンディションに含まる状態遷移以外は、サブタイプが保証する必要はない

モデリングの原則

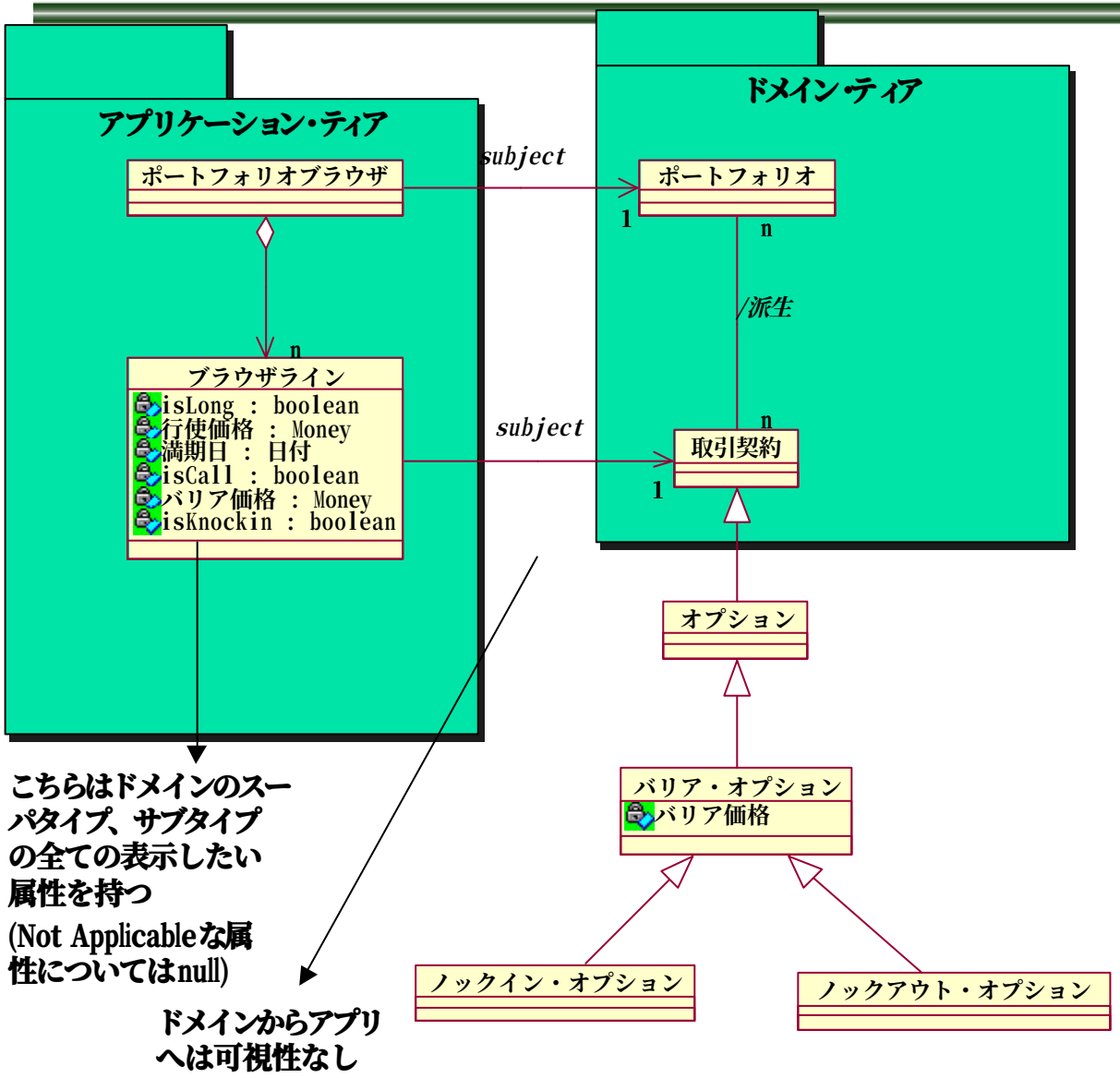
- 状態図における汎化の効果はよく理解されていない。スーパタイプ上の全イベントが、サブタイプで扱えることを確認することは重要である。サブタイプ化される状態図は、必ず未知のイベントを許すべきである（矢崎、この部分？）

- 事後条件とは、その操作の後にオブジェクトにおいて真となるべき条件であると定義される。事後条件に言及されない他の変化が起きてもよい

ちなみに

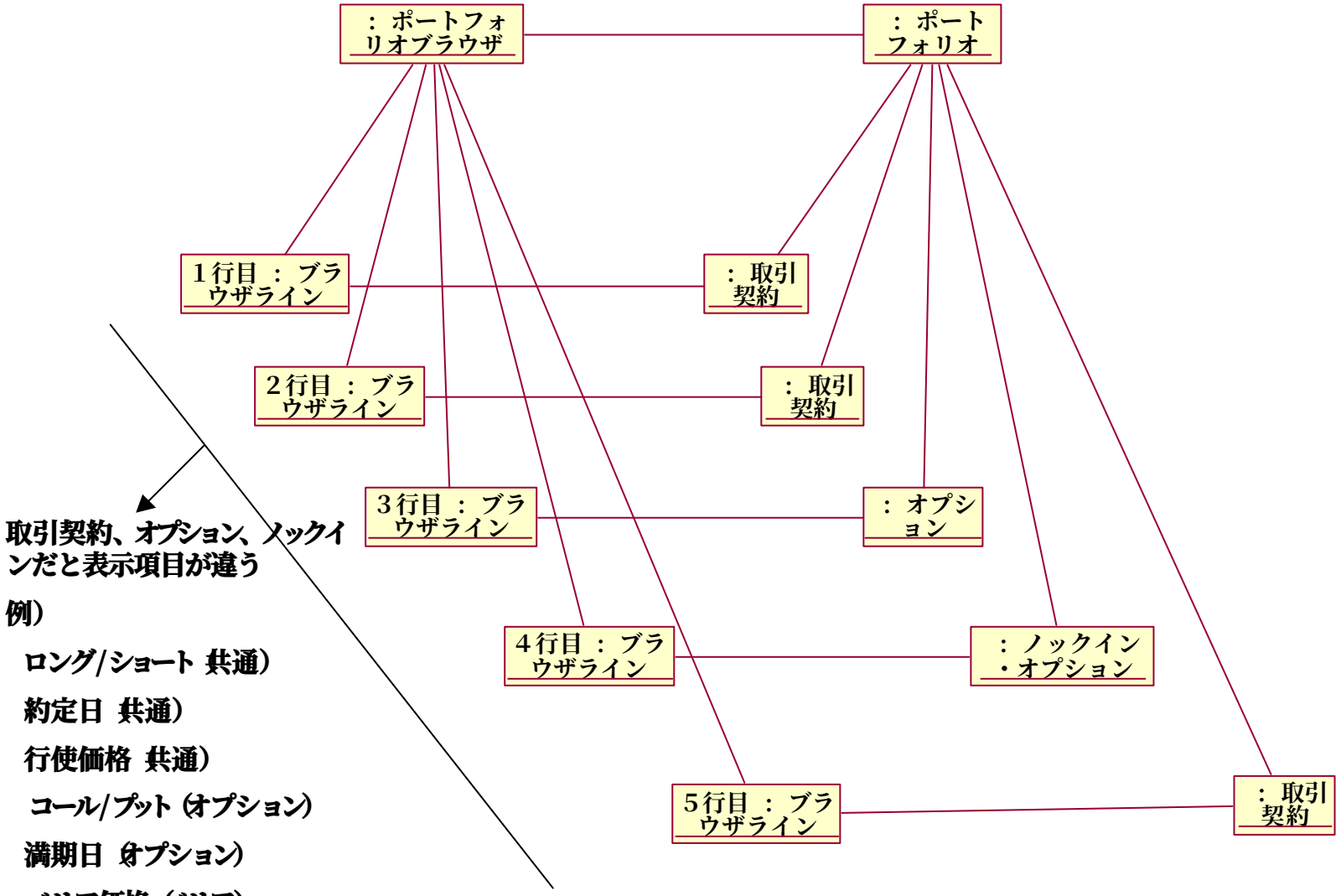
- P.351 翻訳本)のパターン一覧を確認してみよう
  - パターン名
    - サブタイプ状態機械 (Subtype State Machine)
  - 問題
    - バリアオプションはオプション取引とは異なる振る舞いを持つが、サブタイプのように見える。サブタイプと状態機械を扱う。
  - 解決
    - サブタイプとスーパタイプのオブジェクトの両方が、同じ事象に反応することを保証する。

並行なアプリケーション階層とドメイン階層



- ユーザインタフェース上にオブジェクトのリストを表示している。このオブジェクトはさまざまなサブタイプであり、いくつかのサブタイプの属性を表示する必要がある。このユーザインタフェースオブジェクトは、不適当なオブジェクトにメッセージを送ることによって失敗してはならない

並行なアプリケーション階層とドメイン階層



よくわからん

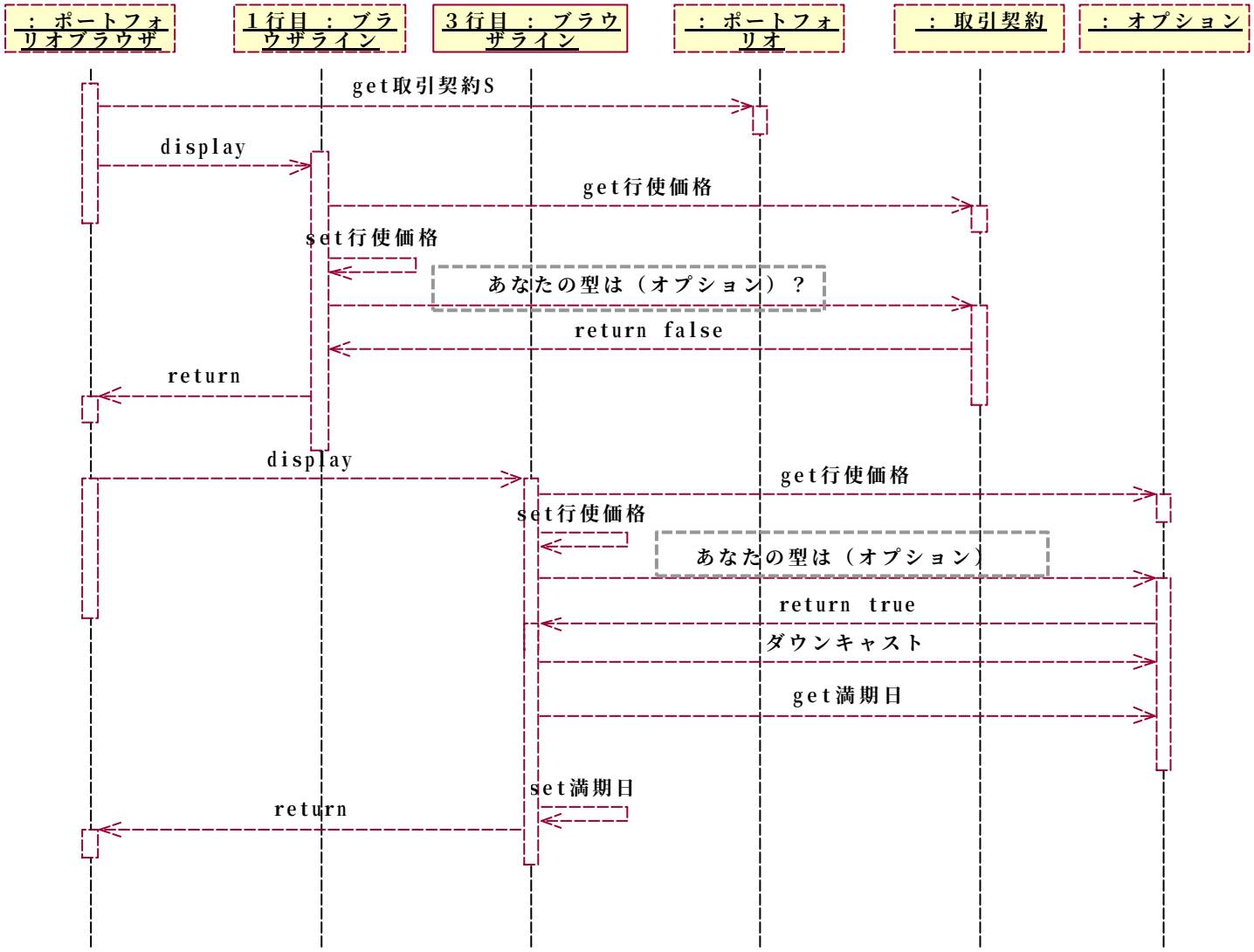
- P. 229
  - ブラウザラインのクラスが、各取引契約に固有に関係する属性を要求するようにしてはいけない。このようなアプローチだと、ブラウザラインのクラスは、非オプション取引に対して満期日を要求できないのでうまく機能しない

    - What we cannot do is assume that some browser line class asks each contract for each relevant attribute. Such an approach would not work because the browser line class cannot ask a nonoption for its expiration date, since by definition a nonoption does not have one.

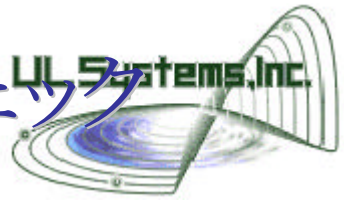
解決方法

- アプリケーションファサードによる型チェック
- スーパタイプにインタフェースを包含させる
- ランタイム属性の使用
- ドメインモデルに対して、アプリケーションファサードを可視にする
- 例外処理を用いる

アプリケーションファサードによる型チェック

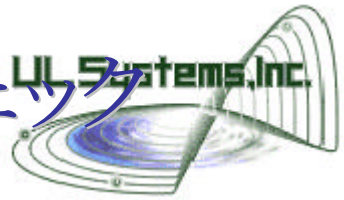


アプリケーションファサードによる型チェック



- ブラウザラインがこの問題に対処する
  - タイプをチェックし
  - ダウンキャストし
  - リクエストを送る

アプリケーションファサードによる型チェック



- 欠点
  - 取引契約のサブタイプが多いと、ブラウザクラスが複雑になる
  - 取引契約の階層構造の変化がブラウザに波及する

- 型チェックを行う程度を軽減させる方法
  - 取引契約の各サブタイプに対応してブラウザラインのサブクラスを用いる方法
    - 作業を行うブラウザラインの正しいサブクラスをインスタンス化するために、型チェックを使う・・・・・？　(よく意味がわからない　矢崎))
  - Visitorパターンを使う方法

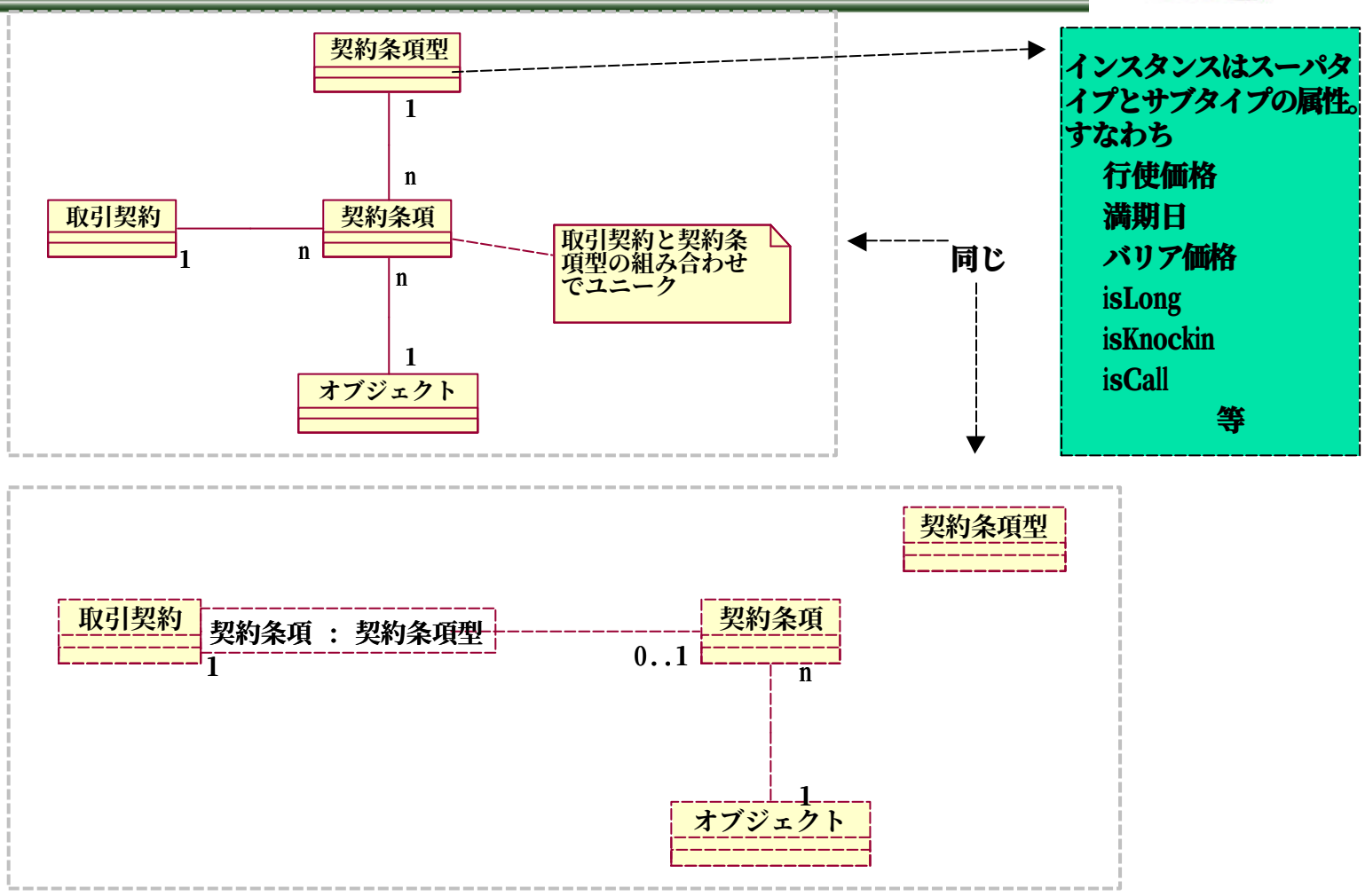
※これらの2つの方法は、ブラウザライン（とそのサブクラス）が、取引契約の階層構造を知っていなければならないというマイナス面もある

スーパタイプにインタフェースを包含させる

- ドメインのスーパタイプに、そのサブタイプのすべての操作を置く
  - スーパタイプではこれらすべてのリターン値はnull
  - 適切な値を返さなければならないサブタイプはオーバライドする

- 欠点
  - スーパタイプの真に正しい処理がどれで、エラーはどれかを区別することが不可能になる
  - サブタイプが導入される都度、スーパタイプのインタフェースを変更しなければならない

ランタイム属性の使用



ランタイム属性の使用

- 取引契約が契約条項を求められたとき、その契約条項があればその値オブジェクトを返す。なくても、エラーとしない
  - nullを返すか、nullオブジェクトを返すか。。そんなところか。

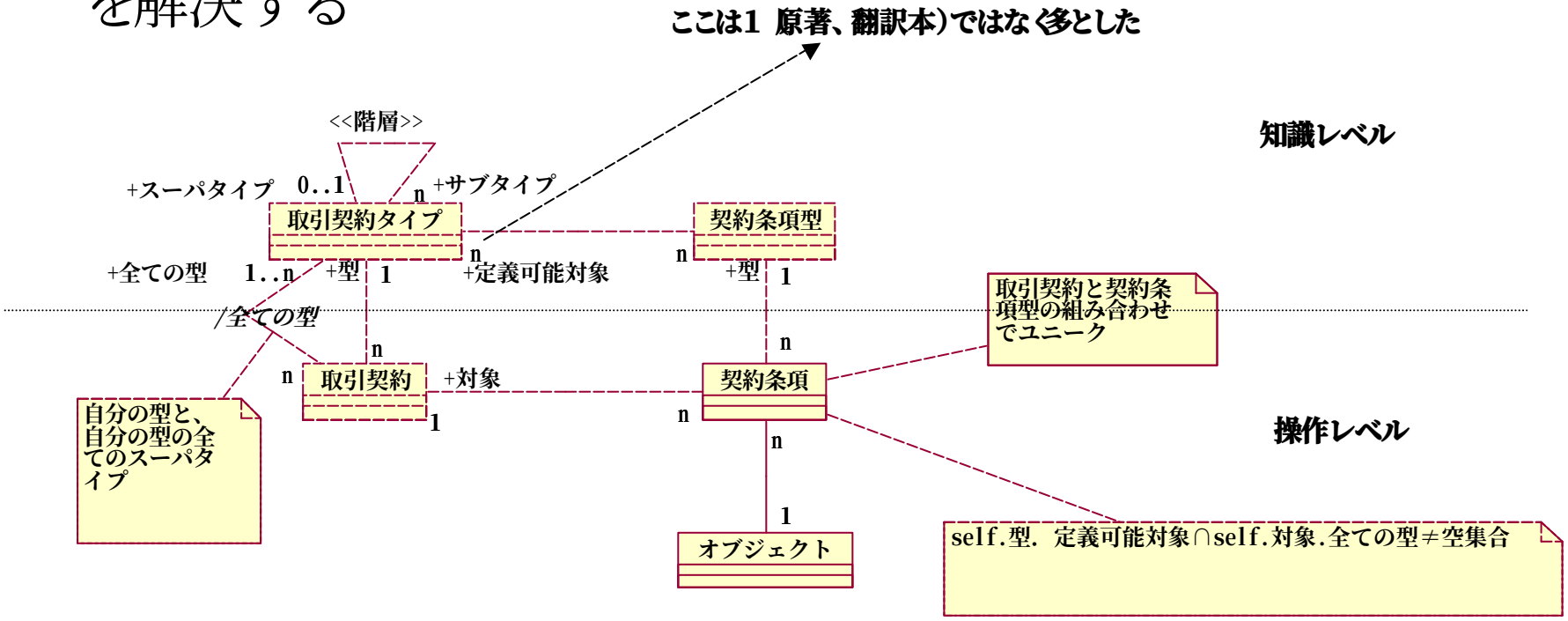
ランタイム属性の使用

- メリット
  - 概念モデルにおいてのメリット
    - 概念モデルの変更なしに、型に属性を追加できる
  - 実装モデルにおいてのメリット
    - システム実行中に、リコンパイルせずに、属性を変更できる

- このモデルの問題
  - この方法だと、もってはいけない属性（取引契約における満期日のような）の値を登録してしまう可能性がある
  - これを回避する方法（2つある）
    - その１:知識レベルを用いる方法
    - その２:契約条項型を導出インタフェースとして扱う方法（矢崎：よくわかりません（T_T））

ランタイム属性の使用

- 知識レベルを用いて、「持ってはいけない属性を持ちうる」問題を解決する

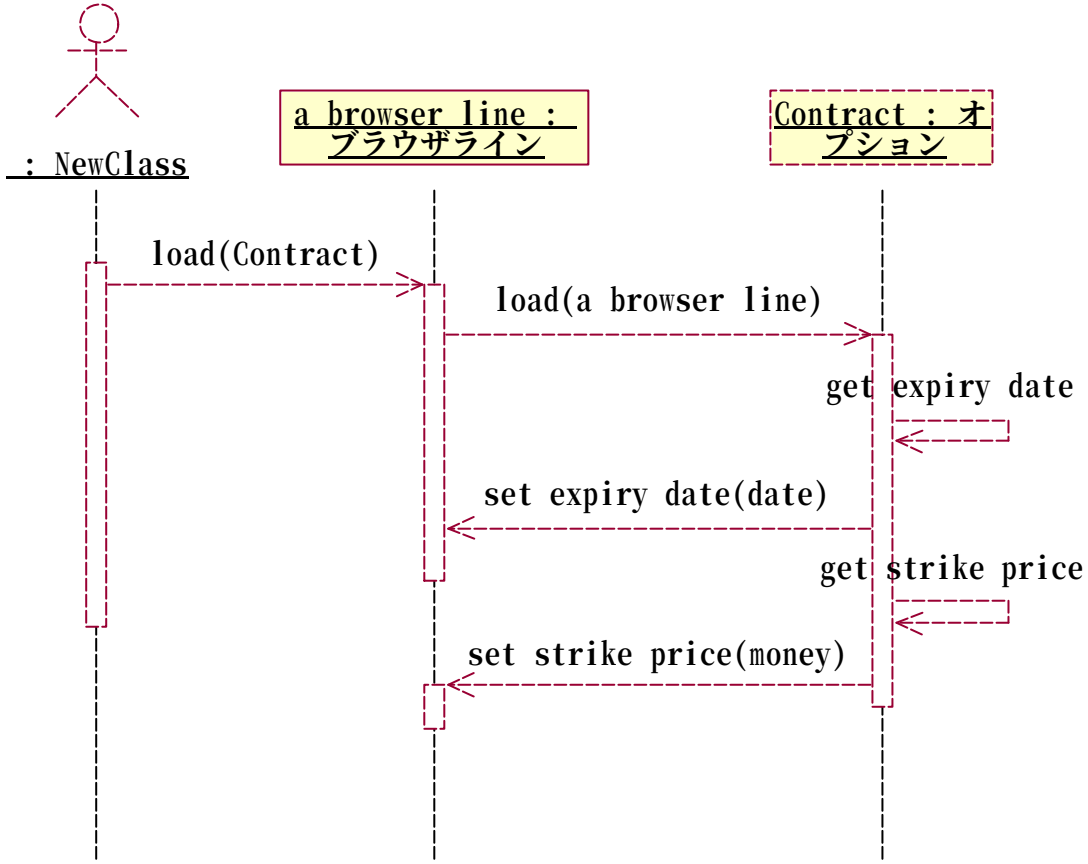


ランタイム属性の使用

- 欠点
  - 取引契約およびそのサブタイプのインタフェースが理解しづらくなる
  - 属性の型について、コンパイル時のチェック（静的な型チェック）ができない
  - 基本的な言語機構が破壊される。コンパイラは何が行われているかに気づかないし、ポリモーフィズムのような言語特徴は、プログラマがハンドコードしなければならない。またパフォーマンスも落ちる

- これらの欠点の多くは、ドメインに両方のインタフェースを設けることで緩和される
  - コンパイル時に属性について知っているソフトウエア部分はモデル属性を使う
  - ブラウザはランタイム属性を使う

ドメインモデルに対して、アプリケーションファサードを可視にする



ドメインモデルに対して、アプリケーションファサードを可視にする

- ブラウザラインは、アプリケーションに必要な情報全てをサポートする

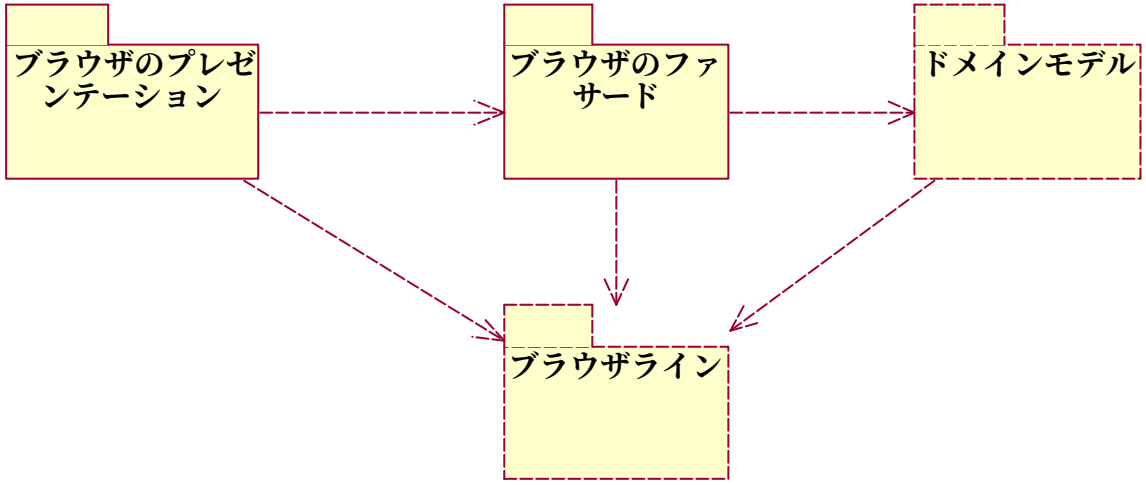
- 取引契約およびそのサブタイプは、自分自身どのような属性を持っているかを知っている

ドメインモデルに対して、アプリケーションファサードを可視にする

- 利点
  - 型チェックが不要なので、インタラクションが単純になる
  - ドメイン側に複雑なインタフェースを設けなくてもよい
  - 新しい取引契約を追加しても、プレゼンテーションに変更がない限り、ブラウザラインの変更は不要
  - 必要なことは、 取引契約のサブタイプの)load操作をオーバライドするだけ
- 欠点
  - アプリケーションティアとドメインティアの間の可視化ルールが破られる（12章参照）

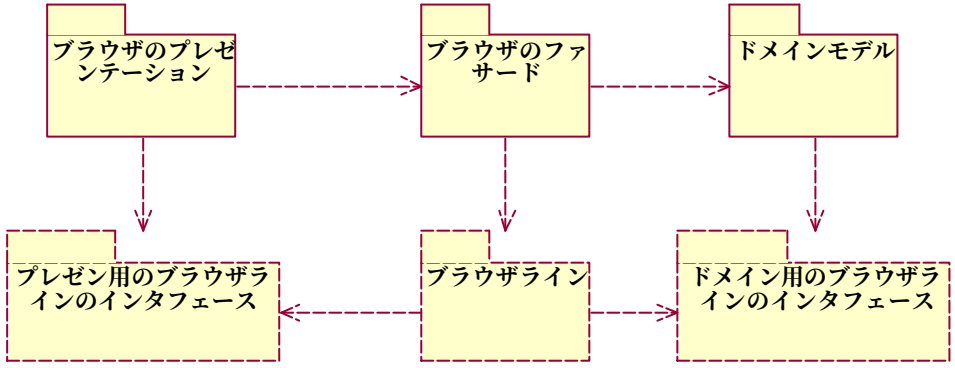
ドメインモデルに対して、アプリケーションファサードを可視にする

- 可視化ルールが破られる点の回避策
  - ブラウザラインを、それ自身のパッケージに配置する
    - この方法を使えば、ドメインモデルの依存先がブラウザラインに限定される

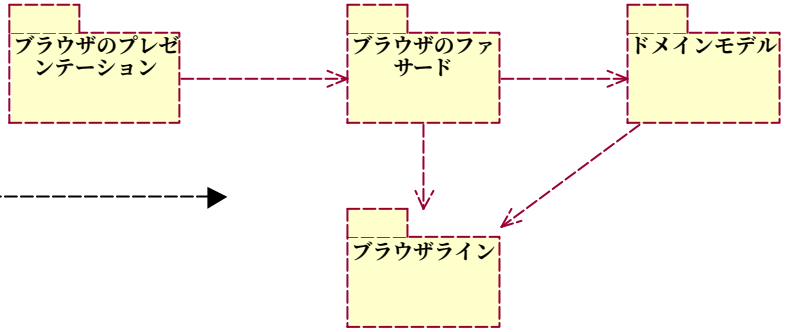


ドメインモデルに対して、アプリケーションファサードを可視にする

- ブラウザライン型を分割すると、可視性はもっと減少できる
  - こんな感じか？

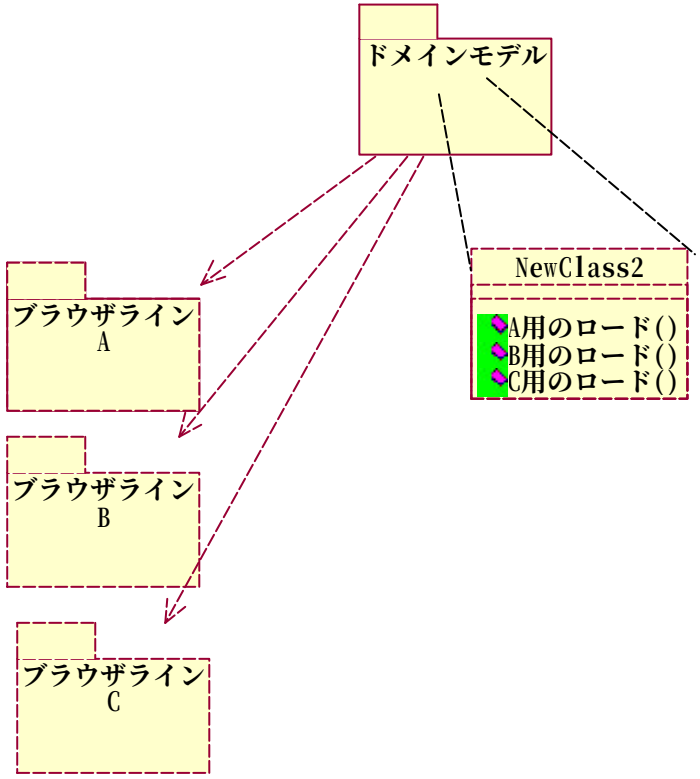


- ファサードを使う方法？



ドメインモデルに対して、アプリケーションファサードを可視にする

- もう1つの欠点
  - 少しずつ異なった必要性を持ったブラウザアプリケーションがいくつもできてしまう場合、ドメインがそれらすべてを知らなければならない



ドメインモデルに対して、アプリケーションファサードを可視にする

- ブラウザラインを分離する方法は、この問題を解決する（ブラウザのファサードが差を処理する方法）
  - 各アプリケーションに対しそれぞれ1つずつブラウザラインのファサードが作られる。
  - それらはすべて同じ1つのブラウザラインパッケージを用いる
  - 新たなブラウザを加えても、ブラウザラインに新しい特性を加えない限り、取引契約の責任が変更されることはない

ドメインモデルに対して、アプリケーションファサードを可視にする

- この方法でも、ブラウザラインに新しい特性を追加したときは、取引契約とそれらのサブタイプの変更が必要になる
  - コントロールをブラウザラインにさせるか、取引契約にさせるかのトレードオフ
  - 新しい取引契約の追加のほうが、ブラウザラインの特性の追加より頻度が高ければ、取引契約にコントロールを置く
  - しかし、可視性のルールを変更する点を軽んじてはならない
  - 取引契約のサブタイプを追加すると、同時にブラウザラインへの特性を追加しなければならないことも多いだろう。したがって、サブタイプの追加がブラウザラインの特性の追加をともなわないケースがそれなりにありえる場合以外は、取引契約にコントロールを置かないほうがよい

例外処理を用いる

- アプリケーションファサードがドメインに処理を要求した結果、例外が返ってきた場合、それを空値として扱う
  - 例）取引契約にバリア価格をリクエストしたら例外が返ってきたので、画面のフィールドに空文字をセットした
  - この場合には、ファサードは、その例外が受け手がその要求を理解できなかったために起こったものであり、他の問題から起きたものでないことをチェックしなければならない
  - この方法は、静的な型チェックを行わない言語を前提としている。ブラウザラインはそれが何の型かをチェックすることなく、全ての要求を出す。