



モデルを使ってソフトウェアを開発しよう

UML初学者向けチュートリアル

株式会社チェンジビジョン

バージョン pdf_0238, 2023-05-31

目次

はじめに	1
この文書の目的	1
対象・目標	1
アイコンの説明	2
注意事項	3
著作権表示	3
諸注意	3
商標等について	3
1 準備	5
1.1 モデリングツールを用意する	5
1.2 実装環境を用意する	5
1.3 演習用のプロジェクトを用意する	6
1.4 ボウリングのスコアのつけ方	8
2 スコアシートの構造を調べる	13
2.1 スコアシートの例	13
2.2 スコアシートの構造をオブジェクト図で表す	14
2.3 スコアシートの構造をクラス図で表す	32
2.4 まとめ	39
3 モデルとコードの対応づけ	41
3.1 対応づけ検討用モデルの作成	41
3.2 変換ルールを使ったコードを確認する	62
3.3 まとめ	66
4 スコアやフレームの状態を調べる	67
4.1 フレームの状態について検討する	67
4.2 フレームの状態をステートマシン図で表す	70
4.3 サービスフレームの扱いについて検討する	76
4.4 スコアの状態をステートマシン図で表す	78
4.5 ゲームの進行について検討する	88
4.6 まとめ	93
5 できあがったプログラムを試す	95
5.1 プログラムを完成させる	95
5.2 プログラムをテストする	102
5.3 まとめ	104

6 まとめ	105
7 他の図、他の機能など	107
付録A: モデルやプログラムの作成例	109
参考文献	111

はじめに

この文書の目的

この文書は、ソフトウェアの開発にモデリングツールを使う方法などについて独習するためのチュートリアルです。

このチュートリアルは、ソフトウェアの開発に、UMLを使って描いたモデルが役に立つことを実感する機会を提供することを目的としています。主に、これからソフトウェア開発にモデル図を使うことについて学ぶみなさんを対象にしています。また、モデリングツールを使うことにも慣れてもらえるよう、「astah* Professional」使って演習します。

対象・目標

チュートリアルの対象者

このチュートリアルは、次のような方を対象者と想定しています。

チュートリアルの対象者

- ・ モデリングやUMLについてこれから学ぼうとしているみなさん
- ・ ソフトウェア開発工程は知っているが、まだモデル図を活用していないみなさん
- ・ 分析・設計・テストなどソフトウェアの開発に携わっているみなさん

チュートリアルの目標

みなさんがチュートリアル終了後に次のような人になっていることを、このチュートリアルの目標としています。

チュートリアルの目標(人物像)

- ・ 開発工程のどこでどのモデル図を使えばよいか知っている(知っている人になる)。
- ・ 「開発にはモデルがあったほうがいい」と考えている。
- ・ 自分だけでなく他の人にもモデリングを勧めたいと考えている。

アイコンの説明

本文中では、次のようなアイコンを用いています。



告知。みんなが覚えておくとよい追加情報など。



ティップス。覚えておくと便利なちょっとしたヒントやコツ。



重要。間違えたり見落としたりすると期待通りの結果が得られない設定や操作など。



注意。気をつけないと問題の発生につながるようなことから。



警告。守らないと破損や怪我などにつながる可能性のあることから。

注意事項

著作権表示

©2023 by Change Vision Inc.

「モデルを使ってソフトウェアを開発しよう: UML初学者向けチュートリアル」

本文書ならびに関連する成果物の著作権は、株式会社チェンジビジョンが保有します。



本文書ならびに関連する成果物は、クリエイティブ・コモンズ <表示—非営利—改変禁止 4.0> ライセンスに則り提供します。

クリエイティブ・コモンズ<表示—非営利—改変禁止 4.0ライセンス>

<https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode.ja>

諸注意

- 本文書は、株式会社チェンジビジョン(作成者)が編集したもので、本文書に関する権利、責任は作成者が保有します。
- 本文書に記載されている情報は、本文書を更新した時点のものであり、利用時にはURLなどの各種の情報が変更されている可能性があります。
- 本文書は演習用テキストとしての使用を想定し、内容等は予告なしに変更することがあります。また、作成者がその内容を保証するものではありません。
- 本文書の内容に誤りや不正確な記述がある場合も、作成者は一切の責任を負いません。
- 本文書に記載の内容や実施結果からいかなる損害が生じても、作成者は責任を負いかねますので、あらかじめご了承ください。
- 本文書の複製、保存については、作成者の同意を得てください。

商標等について

- astahは、株式会社チェンジビジョンの日本、米国、欧州における登録商標です。
- Windows 並びに同社の各製品名は米国マイクロソフトコーポレーションの米国およびその他の国における登録商標です。
- Apple、iCloud、iPad、iPhone、Mac、Macintosh、macOSは、米国およびその他の国々で登録されたApple Inc.の商標です。

- ・ Linuxは Linus Torvalds氏、米国及びその他の国における登録商標 あるいは 商標です。
- ・ UNIXは、X/Open Company Limitedが独占的にライセンスしている米国ならびに他の国における登録商標です。
- ・ その他、本書に記載されている社名、製品名、ブランド名、システム名などは、一般に商標または登録商標でそれぞれ帰属者の所有物です。
- ・ 本文中では © 、® 、™ は表示していません。

1 準備

このチュートリアルでは、モデリングツールとして astah* Professional を、実装には Ruby を使います。最初に、演習に必要な環境を用意しましょう。

1.1 モデリングツールを用意する

このチュートリアルで使用するモデリングツール astah* Professional を用意しましょう。

購入前に試用したい場合は、次のページを参照してください。

astah*製品の評価手順

<https://astah.change-vision.com/ja/shopping/evaluate.html>

購入する場合は、次のページを参照して、自分に合ったライセンスを選択してください。

最適なライセンスを見つける

<https://astah.change-vision.com/ja/shopping/price.html>

いずれかの方法で、astah* Professional を入手できたら、次のページを参照して、自分に合ったライセンスを登録してください。

ライセンス登録方法

<https://astah.change-vision.com/ja/registry.html>

1.2 実装環境を用意する

このチュートリアルでは、実装に Ruby を使います。執筆時点では 2.7.2 を使いました。

Ruby は、Windows、Mac、Linux で動作するオープンソースの言語プログラミング言語です。インストール方法は利用環境によって異なります。詳しくは、Ruby の公式サイトの説明や、関連するウェブサイトの記事を参考してください。

Rubyのインストール

<https://www.ruby-lang.org/ja/documentation/installation/>

このチュートリアルでは、Ruby の高度な機能、規模の大きなライブラリやフレームワークは使いません。ですが、Rubyについて多少の知識は必要になるでしょう。Rubyになじみがない人は、公式サイトにある次のようなページを参考になるとよいでしょう。

他言語からのRuby入門

<https://www.ruby-lang.org/ja/documentation/ruby-from-other-languages/>

20分ではじめるRuby

<https://www.ruby-lang.org/ja/documentation/quickstart/>

1.3 演習用のプロジェクトを用意する

業務やシステムを分析(あるいは設計)するときにUMLのモデル図を使う場合、モデリングの対象のいくつかの側面について、それぞれに向いた図を作成することが一般的です。

複数の図で表すのは、大きな問題ならば分割して小さな問題として捉えるほうが整理しやすいからです。たとえば、システム全体の構造と、その内部のある構成要素の内部の構造を別の図に分けることができるでしょう。また、関心事によって表すもの(モデルに残すものや削るもの)が変わるからです。たとえば、あるシステムを設計するのに、構造を表すためにクラス図を使い、振舞いを表すのにシーケンス図やステートマシン図使うといったことです。

astah* でも、モデリングの対象に関連した複数の図をまとめて扱えるよう、ひとまとめのファイルに保存します。astah* では、この複数の図のまとめのことを「プロジェクト」と呼んでいます。

1.3.1 astah* のプロジェクトを作成する

それでは、演習用に astah* のプロジェクトを用意しましょう。

- ・ astah* を起動する。
- ・ 「ファイル」メニューから「プロジェクトの新規作成」を選択する(図 1.1)。

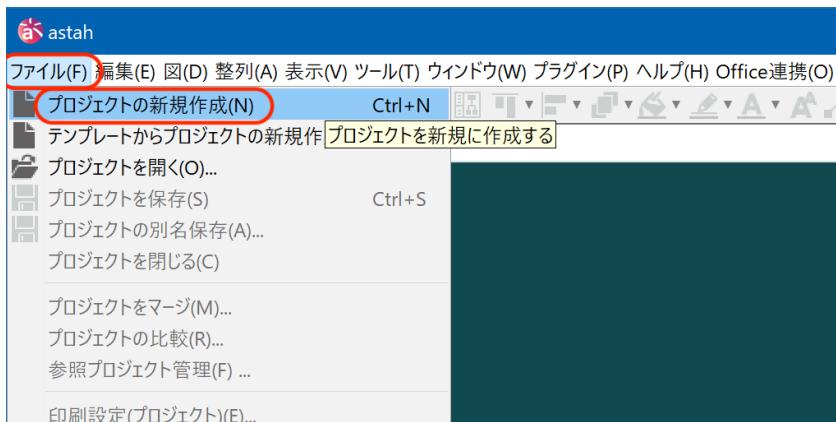


図 1.1 新しいプロジェクトを作成する

- ・ 「no_title」という名前で新しいプロジェクトが作成される(図 1.2)。

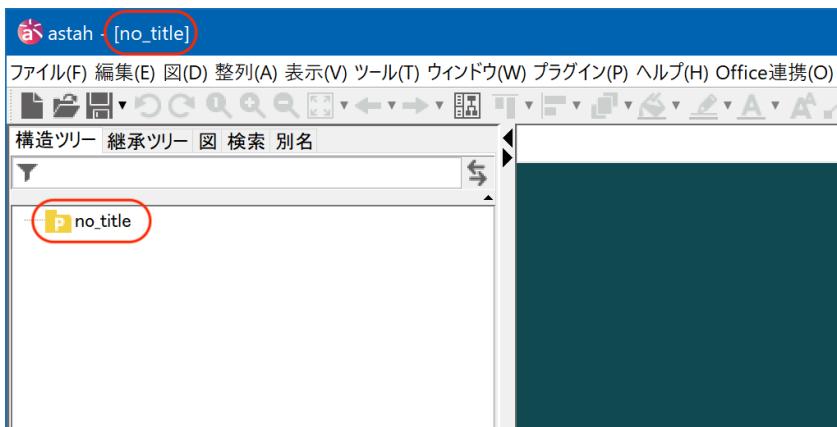


図 1.2 新しいプロジェクトが作成された

1.3.2 作成したプロジェクトを保存する

新規プロジェクトを作成したら、最初に保存しておき、それからモデルの作成に入るようにするとよいでしょう。

- 「ファイル」メニューから「プロジェクトを保存」を選択する(図 1.3)。

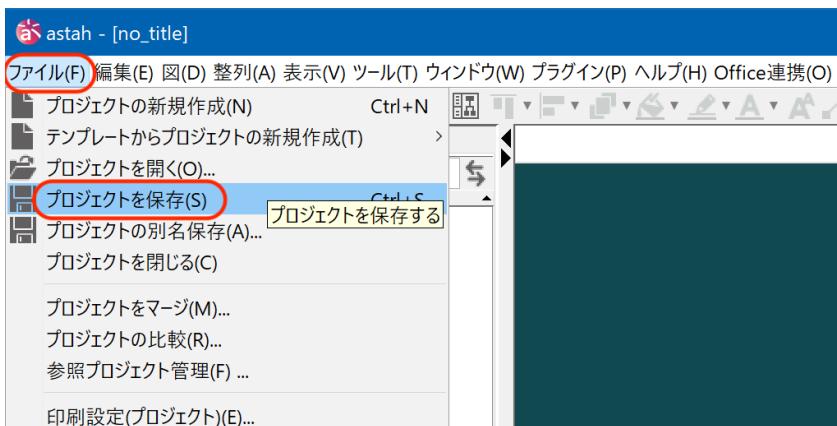


図 1.3 新しいプロジェクトを保存する

- 「保存」ダイアログが開くので、プロジェクトの保存場所と保存するファイル名を指定する。
 - ここでは、保存場所として「デスクトップ」に「BowlingScore」フォルダーを作成した。
 - ファイル名に「bowling_score」を指定して保存した(図 1.4)。

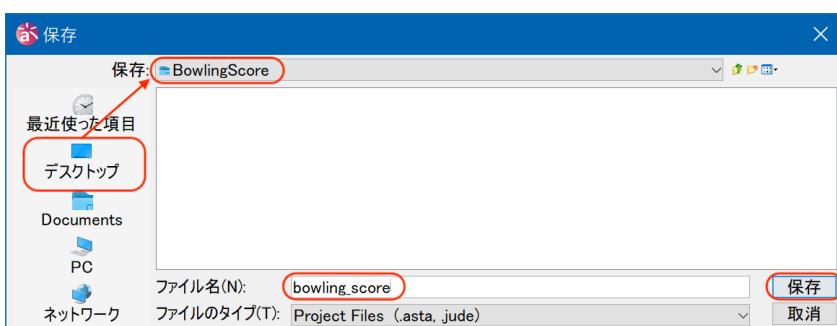


図 1.4 ダイアログを操作してプロジェクトを保存する場所とプロジェクトファイル名を指定する

- 保存すると、プロジェクトファイルにつけた名前が「構造ツリー」に反映される(図 1.5)。

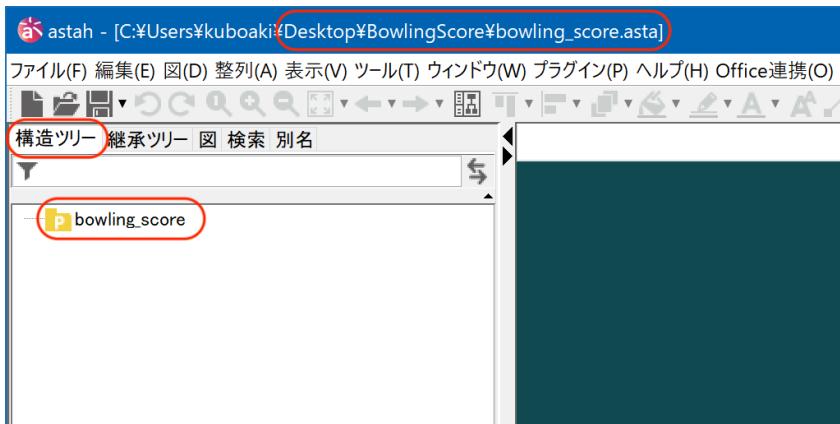


図 1.5 プロジェクトファイル名が、プロジェクト名に反映された



astah* のプロジェクトの作り方についての詳しい説明は、機能ガイドの「プロジェクトの作成と利用」を参考にしてください。

astah* 機能ガイド プロジェクトの作成と利用

<https://astah.change-vision.com/ja/manual/278-project-file.html>

1.4 ボウリングのスコアのつけ方

ボウリング場では、だいぶ前(1980年代の終りくらい)から、スコアを自動計算してディスプレイに表示するシステムが使われるようになっています。自動化される前は、受付でスコアシートをもらって、自分たちでスコアを計算し、手書きでスコアをつけていました(図 1.6)。



図 1.6 手でスコアを記入している様子

ボウリングの規則とスコアのつけ方について、公益財団法人「全日本ボウリング協会」の「ボウリング競技規則」を参考に、まとめてみました。

公益財団法人「全日本ボウリング協会」のボウリング競技規則

<https://jbc-iwate.com/service/message/201012101.pdf>

1.4.1 ボウリングスコアに使う記号

ボウリングスコアに使う記号と意味を 表 1.1 に示します。

表 1.1 ボウリングスコアに使う記号と意味

記号	名称	意味
	ストライク	1投目で10本全てを倒した場合。ストライクをだしたフレームの得点は、次の2投分を加算する。2回続いたら3フレーム目の1投目までを加算する。スコア欄は、それまでは空欄にしておく。ストライクが続くことを「ダブル」「トリプル(ターキー)」と呼ぶ。
	スペア	1投目で残ったピンを、2投目で全部倒した場合。次の1投分を加算する。スコア欄は、それまでは空欄にしておく。
	ミス	フレームの2投目でピンを1本も倒せなかった場合。2投目でガターに落ちても「G」の記号は使わずにこの記号を使う。そのフレームの得点は、1投目で倒したピン数になる。なお、1つのフレームで2回投球し、10本のピンを全部倒すことができなかった場合をエラーという。
	ガター	1投目でレーンの両脇にある溝(ガター)にボールが落ちた場合。2投目はガターに落ちた場合も「ミス」として扱う。
	スプリット	1投目で1番ピン(ヘッドピンともいう)と他のいくつかのピンが倒れ、2本以上のピンが次のような状態に残った場合。スプリットの場合は、1投目のピン数を○で囲む。(残っているピンの中間のピンが少なくとも1本倒れたとき。例えば7と9あるいは3と10。残っているピンのすぐ前のピンが少なくとも1本倒れたとき。例えば5と6。)
	ファウル	ファウルラインを越えて投球されたという記号。1投目でファウルした場合、ピンを倒しても1投目のピン数は0になる。2投目は全ピンを立て直す。2投目でファウルした場合、ピンを倒しても2投目のピン数は0になる。第10フレームの3投目がファウルの場合、ピンを倒しても3投目のピン数は0になる。

1.4.2 クラシックスコアリング

古くから使われていて、広く普及しているスコア計算方式です。

クラシックスコアリングにおけるスコア計算の規定

- ボウリングの1ゲームは、10個のフレームをもって構成する。
- 競技者は、ストライクの場合を除き、それぞれのフレームで2回ずつ投球する。
 - ただし、第10フレームがストライクまたはスペアの場合、サービスフレームが追加される。
 - ストライクの場合は、サービスフレームで2回投球できる。
 - スペアの場合は、サービスフレームで1回投球できる。
- ゲームの成績は、10個のフレームの合計点によってこれを表す。
 - 適正な投球によって倒されたピンの数をもって計算する。
 - 適正に投球されたボールとは、競技者を持っているボールが、手から放れファウルラインを越えたものをいう。

名前	1	2	3	4	5	6	7	8	9	10	Total	
くぼあき	6 9	3 18	9 —	G 21	3 38	7 58	△ 78	9 96	(8) F 104	△ 130	6 150	150/150

図 1.7 スコアの例(クラシックスコアリングの場合)

1.4.3 カレントフレームスコアリング

2018年アジア競技大会で採用された新しいスコア計算方式です。

クラシックスコアリングと異なる点

- 1投目によって10ピンすべてを倒した場合はストライクとなる。ストライクのとき、そのフレームの得点は30となる。
- 続けて2回ストライクの場合は、それぞれのストライクの得点は30となる。
- 1投目の後、残ったピンを2投目によって全部倒した場合は、スペアとなる。スペアのとき、そのフレームの得点は、1投目のピン数に10を加えた数となる。
- 第10フレームでストライクあるいはスペアをとった場合も、3投目はない。

名前	1	2	3	4	5	6	7	8	9	10	Total
うえはら	△ 30	△ 60	△ 90	(8) 108	F 8 116	9 F 125	△ 155	8 — 163	G 9 172	△ 202	202/202

図 1.8 スコアの例(カレントフレームスコアリングの場合)

【豆知識】ボウリングの名前の由来

「ボウリングをすること」などを意味する「bowl」という英語は、ラテン語で「泡」や「瘤」を意味する「bulla」に由来する。一方、同じ綴りで食器や容器(ボウル)を意味する「bowl」や「球」を意味する「ball」は、ゲルマン語に由来し、本質的に異なる。

— Wikipedia



ボーリングのハンディキャップは、参加者が普段プレーしているときのアベレージ(平均スコア)を基に、力量に応じて設定するそうです。このチュートリアルでは、ハンディキャップは取り扱わないでおきます。

2 スコアシートの構造を調べる

ボウリングのゲームスコアを記録するスコアシートは、どのような要素で構成されているでしょうか。また、それらの構成要素の間にはどのような関係があるのでしょうか。構成要素や構成要素の関係を表すには「構造のモデル」を使います。構造のモデルを使って、スコアシートの構成要素や、構成要素間の関係を検討してみましょう。



このチュートリアルでは、クラシックスコアリング(1.4.2)の場合について考えることにします。

2.1 スコアシートの例

ゲーム中のスコアシートの例を図2.1に示します。この例では、2人でプレーしています。いまは、2ゲーム目のプレー中で、1人目の6フレーム目の第1投目までゲームが進んでいます。また、彼らは1フレームずつ交代で投球していることがわかります。もし、彼らがもう1ゲーム分プレーする場合には、このスコアの下にもう一度2人分のスコアが追加されるでしょう。

スコアシートのサンプル（クラシックスコアリング）												
プレイ開始 : 2022年02月04日 16時03分 ← プレイ開始日時												
名前	1	2	3	4	5	6	7	8	9	10	Total	
くぼあき	7	—	5	△	△	△	7	5	4	7	5	155
	7		27	52	71	80	100	115	124	141	155	
うえはら	6	3	9	—	G	3	8	7	9	(8)	—	132
	9		18	21		38	58	78	96	104	123	132
名前	1	2	3	4	5	6	7	8	9	10	Total	
くぼあき	6	3	9	—	G	3	8	7	3			51
	9		18	21		38	51					
うえはら	△	—	8	△	7	2	△	△				46
	20		37		46							

図 2.1 スコアシートの例



ボウリングでは、「ゲーム」が、各自が10フレーム分プレーした結果を指す場合と、複数名で10フレーム分プレーしたひと組の結果を指す場合があるようです。これらを呼び分ける方法がわからなかつたので、このチュートリアルでは、前者を「スコア」、後者を「ゲーム」と呼ぶことにします。

実際のボウリング競技において、これらの呼び分方をご存じの方は、その呼び分け方を使うとよいでしょう。

スコアシートを観察してわかるまとめました。

スコアシートを観察してわかること(決めたこと)

- スコアシートには、プレー開始日時が記載されている。
- スコアシートには、プレーヤー名とその人の10フレーム分の投球を記録する欄がある。
 - これを「スコア」と呼ぶことにする。
- 1つのゲームの進行中は、1フレームごとにプレーヤーが交代して投球する。
 - この方式は「ベーカー方式」と呼ばれている。
- スコアシートには、複数人のスコアが記録されている。
 - 同時に進行している複数名のスコアのまとめを「ゲーム」と呼ぶことにする。
- プレーヤーの組を単位として複数回ゲームをプレーできる。

これらを元に、スコアシートの構成要素や構成要素の間の関係をモデル図で表してみましょう。

2.2 スコアシートの構造をオブジェクト図で表す

スコアシートの構造を調べるには、まず実際のスコアシートがどうなっているかを調べるのがよいでしょう。スコアシートに記入されている具体的なデータを使って図を作成するには、オブジェクト図が向いています。

それでは、スコアシートの例を参照しながら、スコアシートのオブジェクト図を作成してみましょう。

2.2.1 プロジェクトに設計モデルを追加する

まず、astah* で作成したプロジェクトに設計モデルを追加しましょう。



「設計モデル」ということばが表すものは、開発に使用する手法や分野によって少しずつ異なっています。このチュートリアルでは、次の2つの側面について表したものを見ると設計モデルと呼ぶことにします。

- 対象とする業務やサービスに必要となる構成要素やその関係を「構造のモデル」(静的モデル)として表したもの。
- 業務やサービスを実現するために必要となる動作を「振る舞いのモデル」(動的モデル)として表したもの。

設計に関わる構成要素は、対象とする業務やサービスを分析して得られることもあれば、開発に採用する資源や方式から得られることもあります。

プロジェクトに設計モデルを追加する

- 構造ツリー上で、プロジェクトを選択する。
- 右クリックしてポップアップメニューを開き、「モデルの追加>モデル」を選択する(図 2.2)。

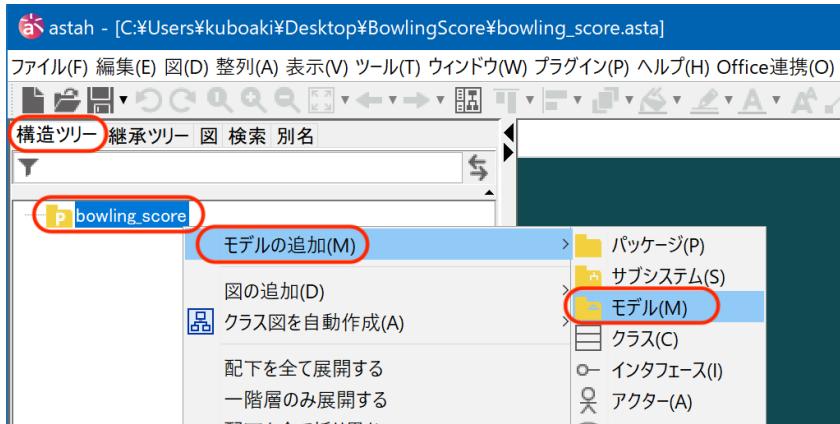


図 2.2 プロジェクトにモデルを追加した

追加したモデルに名前をつける

- 構造ツリー上で、追加したモデルを選択した状態で、プロパティーの「ベース」タブを選択する。
- 「名前」を編集して「設計モデル」とする(図 2.3)。
- 入力が確定すると、構造ツリーの表示にも反映される。

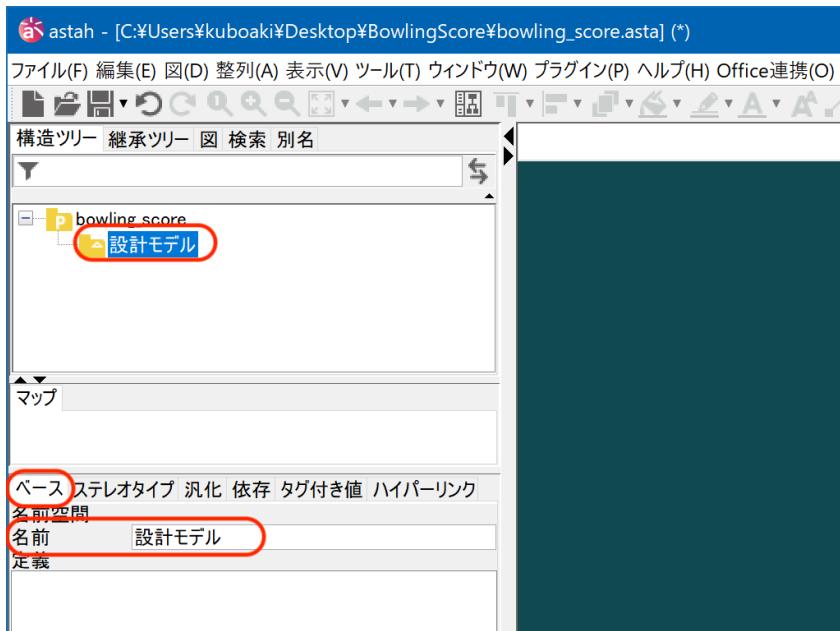


図 2.3 モデルに設計用のモデルとして名前をつけた

2.2.2 設計モデルにオブジェクト図を追加する

「オブジェクト図（インスタンス図と呼ぶこともあります）」を使ってスコアシートに登場するオブジェクトを表してみましょう。astah* でオブジェクト図を作成するときは、「クラス図」を使います。

それでは、プロジェクトにクラス図を追加して、オブジェクト図を描いてみましょう。

設計モデルにオブジェクト図を追加する

まず、プロジェクトにオブジェクト図を描くのに使うクラス図を追加します。

1. 構造ツリーから設計モデルを選択し、右クリックしてポップアップメニューを開く（図 2.4）。
2. 「図の追加>クラス図」でクラス図が追加される。

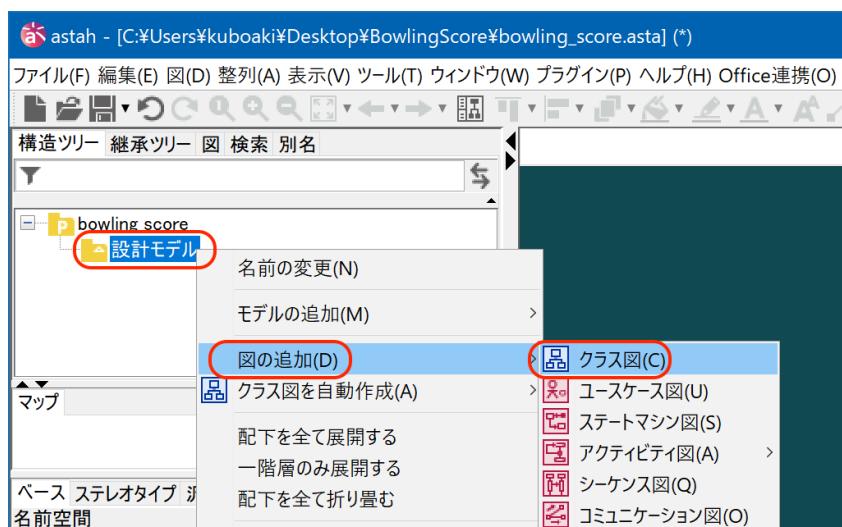


図 2.4 モデルにクラス図を追加する

オブジェクト図に名前をつける

追加したオブジェクト図に、図の名前を設定します。描こうとしているのはオブジェクト図ですので、それがわかるような名前をつけます。

- ・ 追加したクラス図のプロパティーの「ベース」タブを開く。
- ・ 名前を編集して「ゲームスコアのオブジェクト図」とする（図 2.5）。
- ・ ダイアグラムエディタのタイトルやタブにも反映される。

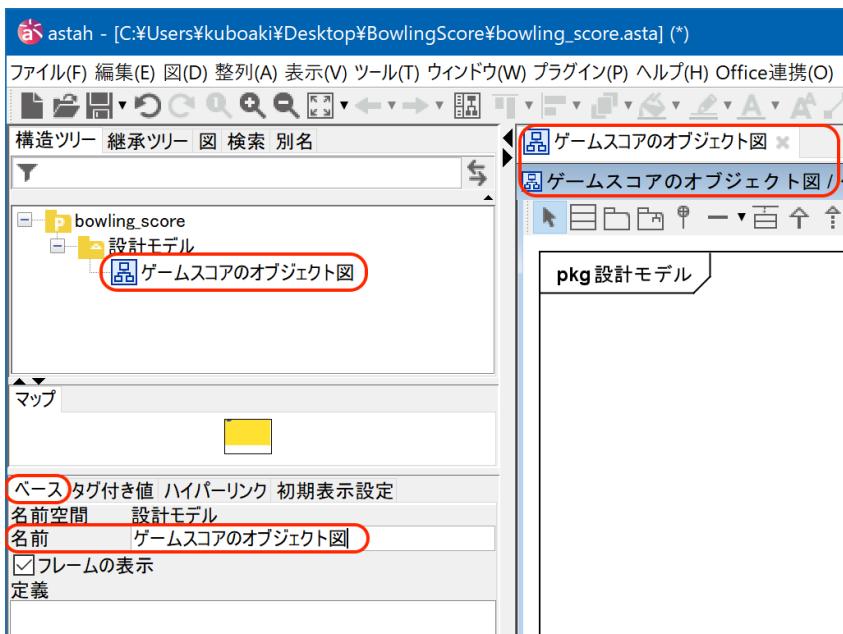


図 2.5 図の名前を「ゲームスコアのオブジェクト図」にする

2.2.3 オブジェクト図にスコアシートを追加する

図 2.1 や「スコアシートを観察してわかること(決めたこと)」を観察しながら、スコアシートに記載されている要素をオブジェクト図に追加してみましょう。まず、「スコアシート」オブジェクトを追加します。

スコアシートを表すオブジェクトを追加する

こんどは、図にスコアシートを表すオブジェクトを追加してみましょう。

- ・ パレットから「インスタンス仕様」を選択して、図に配置する(図 2.6)。
- ・ 「インスタンス仕様0」という名前のオブジェクトが配置される(末尾の数字は作るたびに新しいものに変わる)。

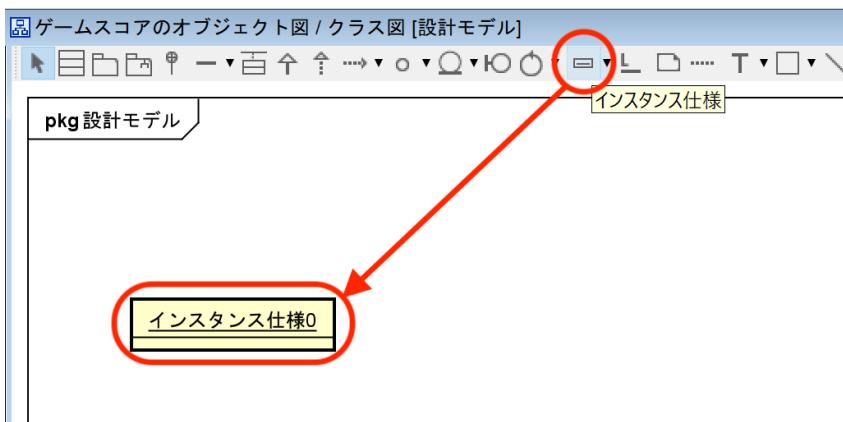


図 2.6 インスタンス仕様を追加する

オブジェクト名を個別のスコアシート名に変える

実際のスコアシートでは、個々のスコアシートを区別するようなはつきりとした名前はついていないこともあります。スコアシート番号みたいな通番をついている場合もあるかもしれません。

ここでは、追加したオブジェクトが特定のスコアシートを表すことがわかるよう、他のシートと区別できるような名前(オブジェクト名)をつけましょう。

- 追加したオブジェクトを選択し、プロパティーから名前の編集欄を選択する(図 2.7)。
- オブジェクトの名前に、図 2.1 のスコアシートを示す固有の入力する。ここでは「scoresheet01」とした。
- オブジェクトの名前の入力を確定すると、ダイアグラムエディタ中のオブジェクトの名前にも反映される。

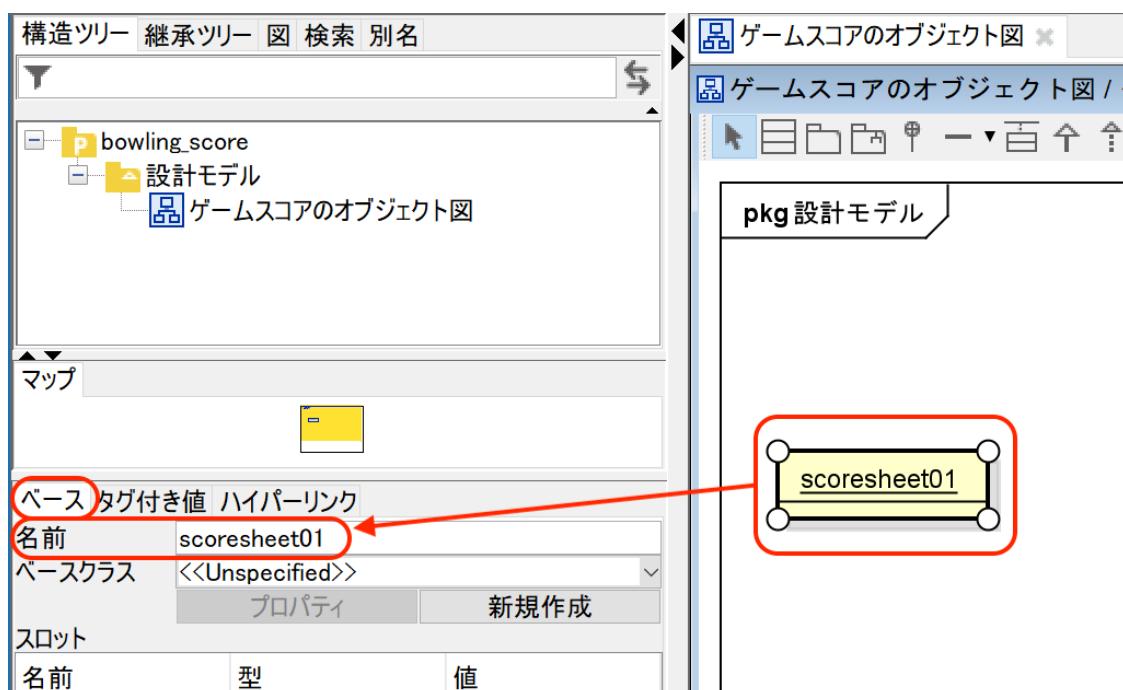


図 2.7 名前を「scoresheet01」に変更する



個々のオブジェクトを識別するためにつける名前のこと、オブジェクト名(またはインスタンス名)と呼びます。

2.2.4 スコアシートオブジェクトにクラスを割り当てる

追加したオブジェクトは、名前はつけたものの、たくさんあるスコアシートの仲間であるとみなす方法はありません。それは、このオブジェクトがどんなクラスに属するか決まっていないからです。それが分かるよう「スコアシート」を表すクラスを定義して割り当てておきましょう。

スコアシートクラスを追加してオブジェクトに割り当てる

まず、図中のオブジェクトに割り当てるクラスを用意します。

- オブジェクト「scoresheet01」を選択した状態で、プロパティーから「新規作成」ボタンをクリックする(図 2.8)。

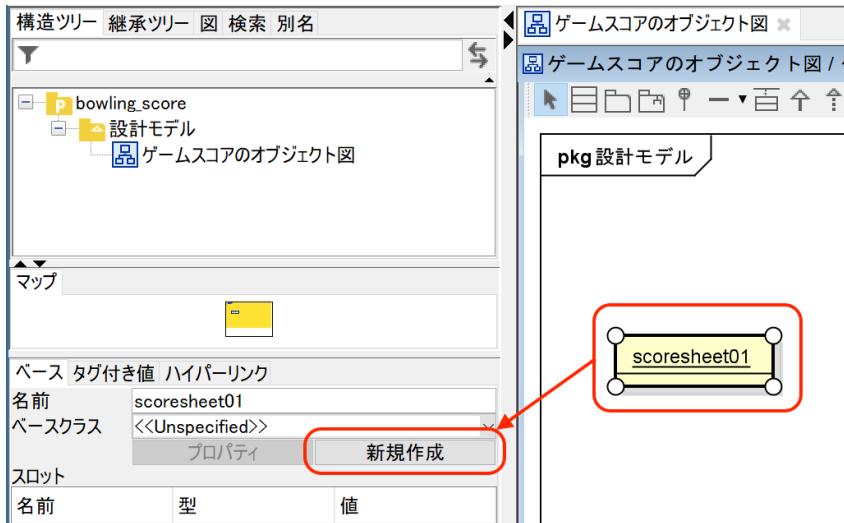


図 2.8 オブジェクトを選択してクラスを「新規作成」する

- クラスを定義するダイアログが表示される(図 2.9)。
- 「ベース」タブを選択し、名前に「ScoreSheet」を入力する。

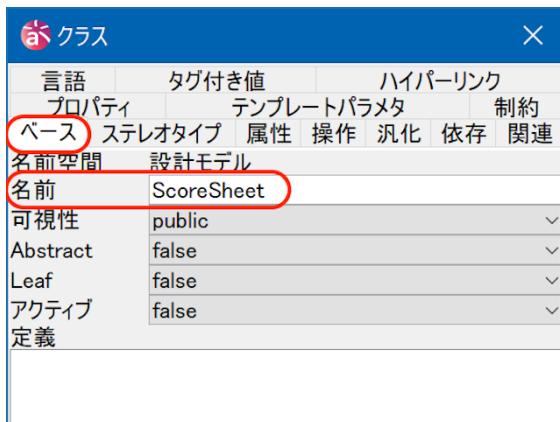


図 2.9 クラス「ScoreSheet」を定義する

スコアシートに属性を追加する

スコアシートクラスがどのような情報を保持するのかはまだ検討が必要ですが、ひとまずプレー開始日時を保持することは必要そうです。そこで、プレー開始日時を表す属性を追加しましょう。ここでは、プレー開始日時を表す属性の名前を「play_date」とします。

- スコアシートクラスの「属性」タブを選択する。

- 「+」ボタンを押すと属性が追加されるので、名前に「play_date」を入力する(図 2.10)。

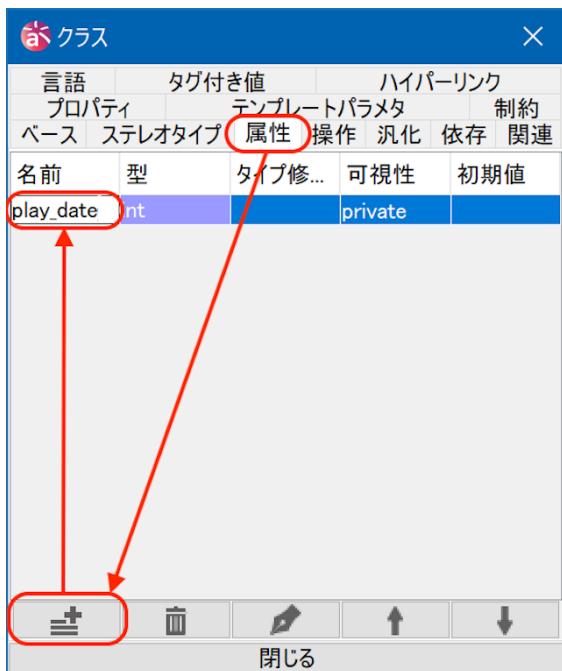


図 2.10 属性「play_date」を追加する

追加した属性の型を定義する

追加した属性も、どんな型なのかを設定しておきましょう。プレー開始日時を表す属性の型の名前を「Time」とします。



「Time」は、Rubyのクラスライブラリのクラスのひとつで、日付と時刻を操作するためのクラスです。このクラスに対応づけする見込みで割当ててみます。

- 追加した属性の「型」欄を選択する。
- 「型」欄を編集状態にして「Time」を入力すると、ダイアログに「型になるTimeを新規作成しますか?」というメッセージが表示される(図 2.11)。
- 「はい」をクリックしてダイアログを閉じる。
- クラス「Time」が作成される。(図 2.12)。
- 構造ツリーにも追加されたことを確認する。

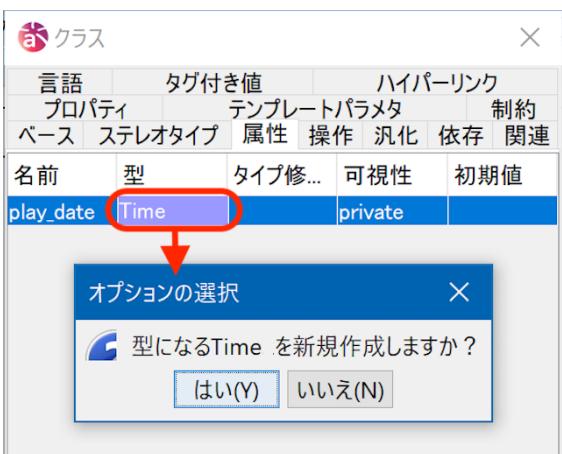


図 2.11 クラス「Time」の追加を促すダイアログ

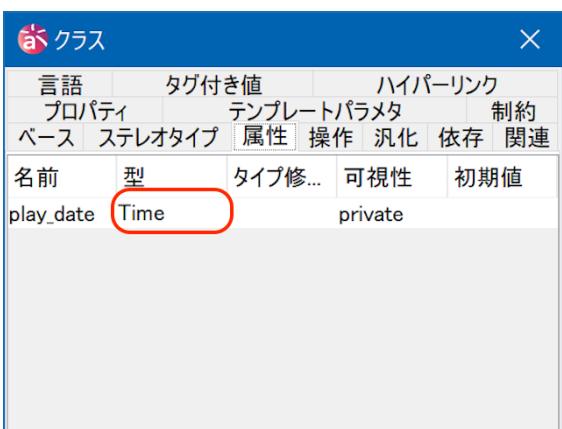


図 2.12 クラス「Time」が追加された

オブジェクト図や構造ツリーを確認する

オブジェクト図においても、構造ツリーにおいても、追加したオブジェクトの名前が変わっているを確認します。

- ・「閉じる」をクリックして、クラス定義のダイアログを閉じる。
- ・オブジェクトの表示が「scoresheet01 : ScoreSheet」に変わっている。
- ・属性「play_date」の属性値を保持する欄(スロットと呼ぶ)も追加されている。
- ・構造ツリーにもTimeクラスが追加されている(図 2.13)。

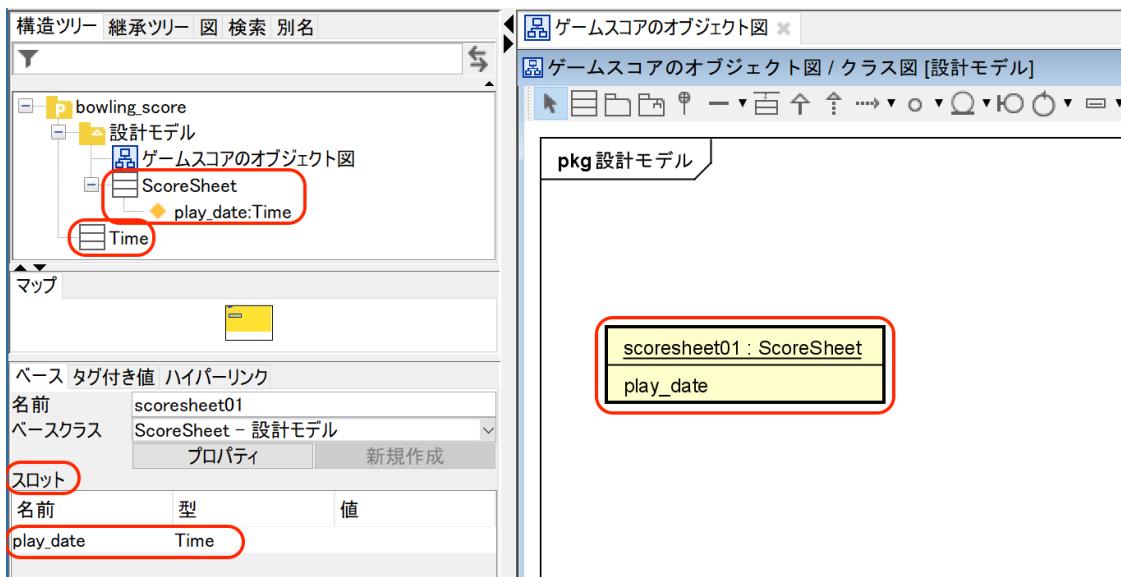


図 2.13 オブジェクトにクラスやスロットが割当てられた

追加したスロットに属性値を設定する

追加した属性「play_date」のインスタンス(スロットと呼びます)に、属性値として具体的な日付データを設定します。

- ・オブジェクト「scoresheet01」を選択して、プロパティから「ベース」タブを開く。
- ・属性「play_date」の値に日時、たとえば「2022/02/04 16:18」などと入力する(図 2.14)。

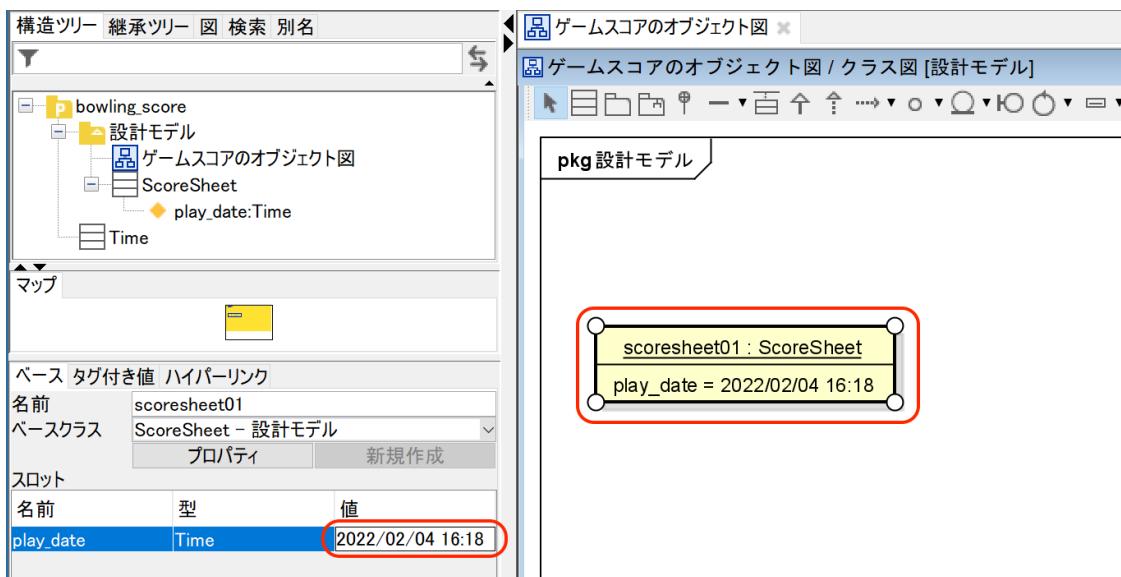


図 2.14 オブジェクトのスロットに属性値が追加された

2.2.5 オブジェクト図にゲームを追加する

次に、「スコアシート」を追加したのと同じ手順で「ゲーム」オブジェクトを追加します。図 2.1 の場合ゲームが2組あるので、ゲームのオブジェクトを「game01」、「game02」としましょう。クラス名は「Game」としましょう。

ゲームを表すオブジェクトを追加する

最初の(1組目)のゲームを追加してみましょう。

- ・ パレットから「インスタンス仕様」を選択して図に配置する。
- ・ オブジェクト名を「game01」とする。
- ・ クラスを追加して「Game」とする(図 2.15)。

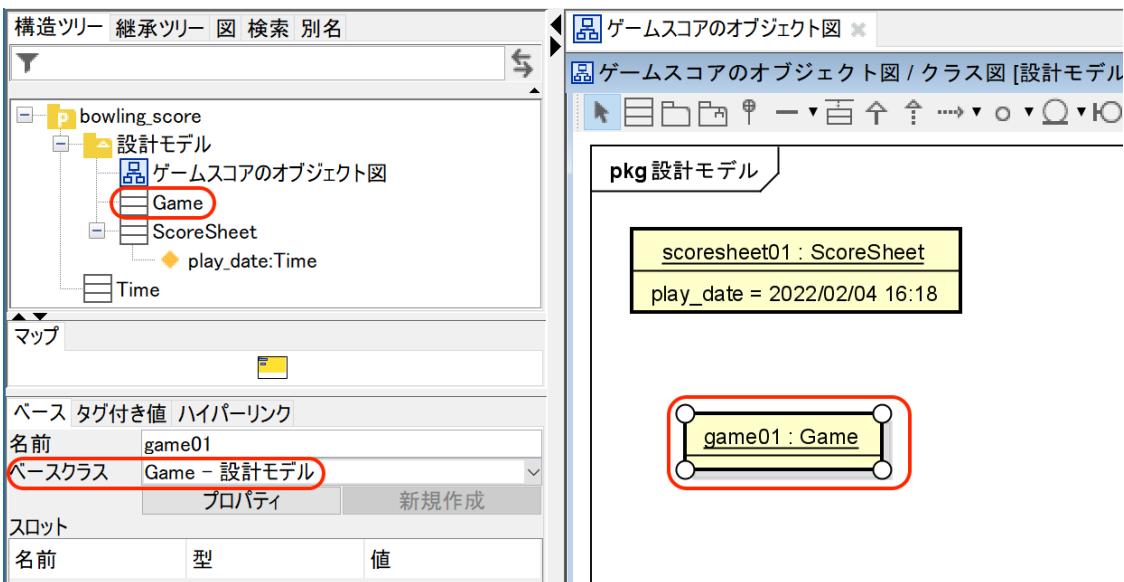


図 2.15 ゲームを表すオブジェクト「game01」とクラス「Game」を追加した

次のゲームのオブジェクトを追加する

次のゲーム(つまり、2組目)のオブジェクトも追加してみましょう。

- ・ 同様の手順で「game02」を追加する。
- ・ プロパティから既存のクラスをプルダウンし、「Game」クラスを選択する(図 2.16)。

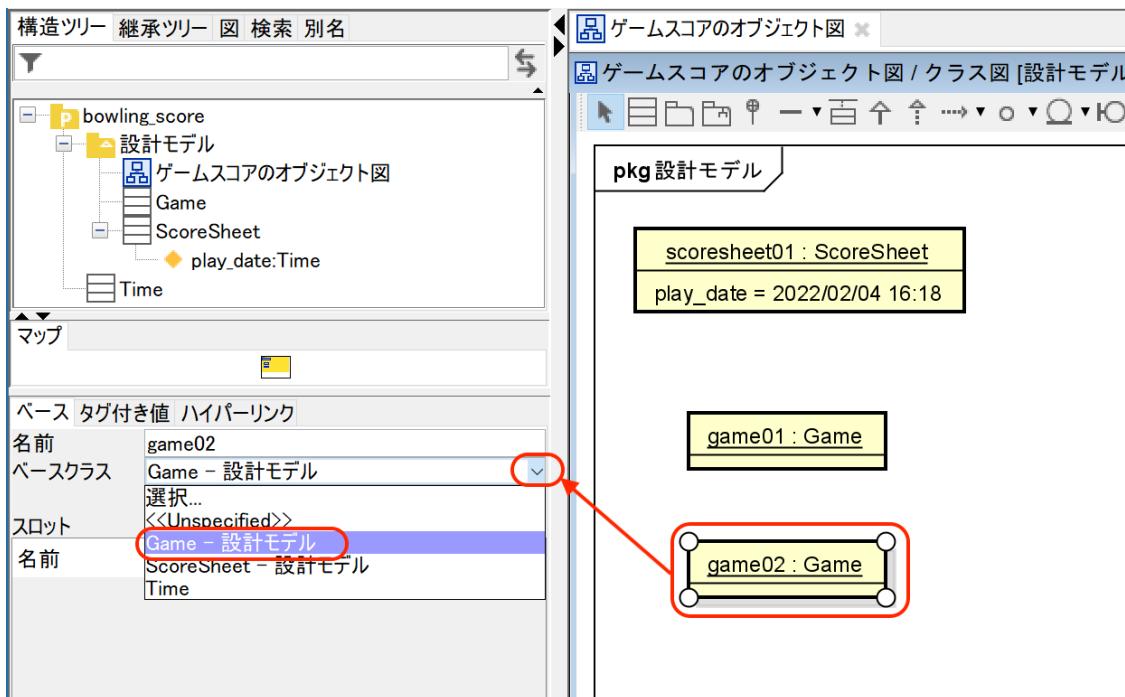


図 2.16 ゲームを表すオブジェクトを追加し、既存のクラスを割当てた

2.2.6 オブジェクト図にスコアを追加する

こんどは、プレーヤーひとり分のスコアを記録しているスコア部分を追加しましょう。図 2.1 の場合、スコアは4つあります。

このチュートリアルでは、プレーヤー名はスコアの属性と考えることにしておきます。(もちろん、プレーヤーを独立したクラスと考え、複数のスコアとを関連づけた方がもっとよいでしょう)

それぞれのプレーヤーのスコアを表すオブジェクトを追加する

それぞれのプレーヤーごとにスコアがありますので、まず、プレーヤーごとのスコアのオブジェクトを追加します。

- ・パレットから「インスタンス仕様」を選択して図に配置して「score01」とする。
- ・「Score」クラスを追加して、「score01」に割当てる(図 2.17)。



図 2.17 スコアを表すオブジェクト「score01」を追加した

- 「Score」クラスに属性「player」を追加し、「String」クラスを追加して割当てる(図 2.18)。

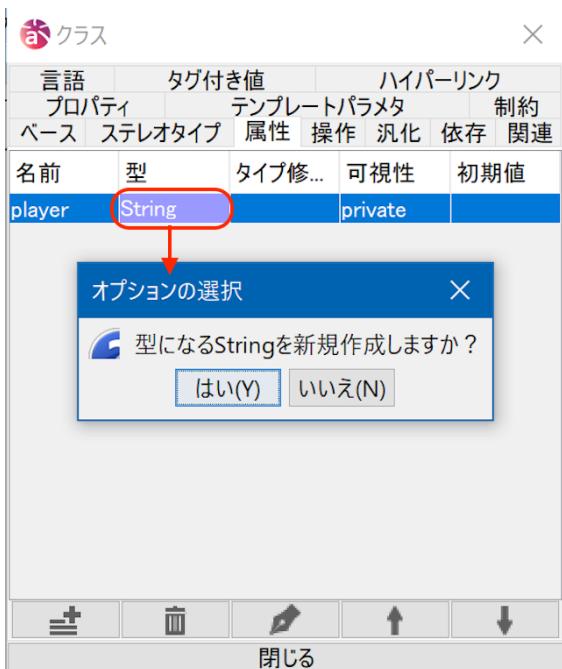


図 2.18 属性「Player」を追加し、クラス「String」を追加して割当てる。

追加したオブジェクトのスロットに値を設定する

追加したオブジェクトの「player」スロット(スコアクラスの属性playerの、このオブジェクトにおける属性値)を設定します。

- 「score01」のスロット「player」の値に「くぼあき」を設定する(図 2.19)。



図 2.19 「player」のスロットの値にプレーヤー名を設定した

- ほかのスコアのオブジェクトも作成する(図 2.20)

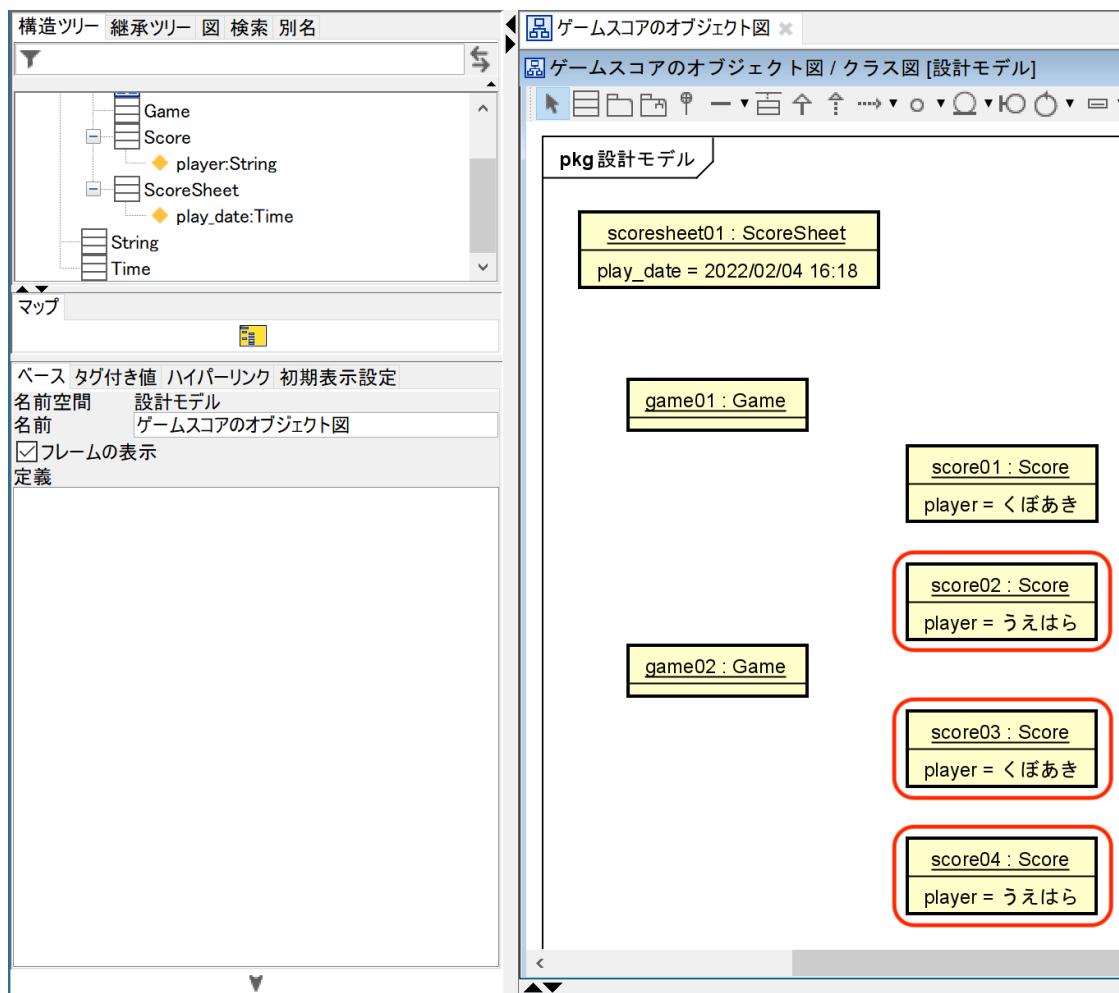


図 2.20 残りのスコアのオブジェクトを作成した

2.2.7 オブジェクト図にフレームを追加する

次に、各スコアに記載されているフレームを追加しましょう。やはり、図 2.1 や「スコアシートを観察してわかること(決めたこと)」を観察しながら、作成します。ですが、フレームのオブジェクトの数が多いので、この場で作成してみるのは一部だけにします。

それぞれのスコアのフレームを表すオブジェクトを追加する

それぞれのスコアは、フレームを持っていますので、これを表すオブジェクトを追加します。

- ・ パレットから「インスタンス仕様」を選択して図に配置して「frame0101」とする。
- ・ 「Frame」クラスを追加して、「frame0101」に割当てる(図 2.21)。
- ・ 「Frame」クラスの属性
に「frame_no」、「first」、「second」、「spare_bonus」、「strike_bonus」、「total」を追加する。

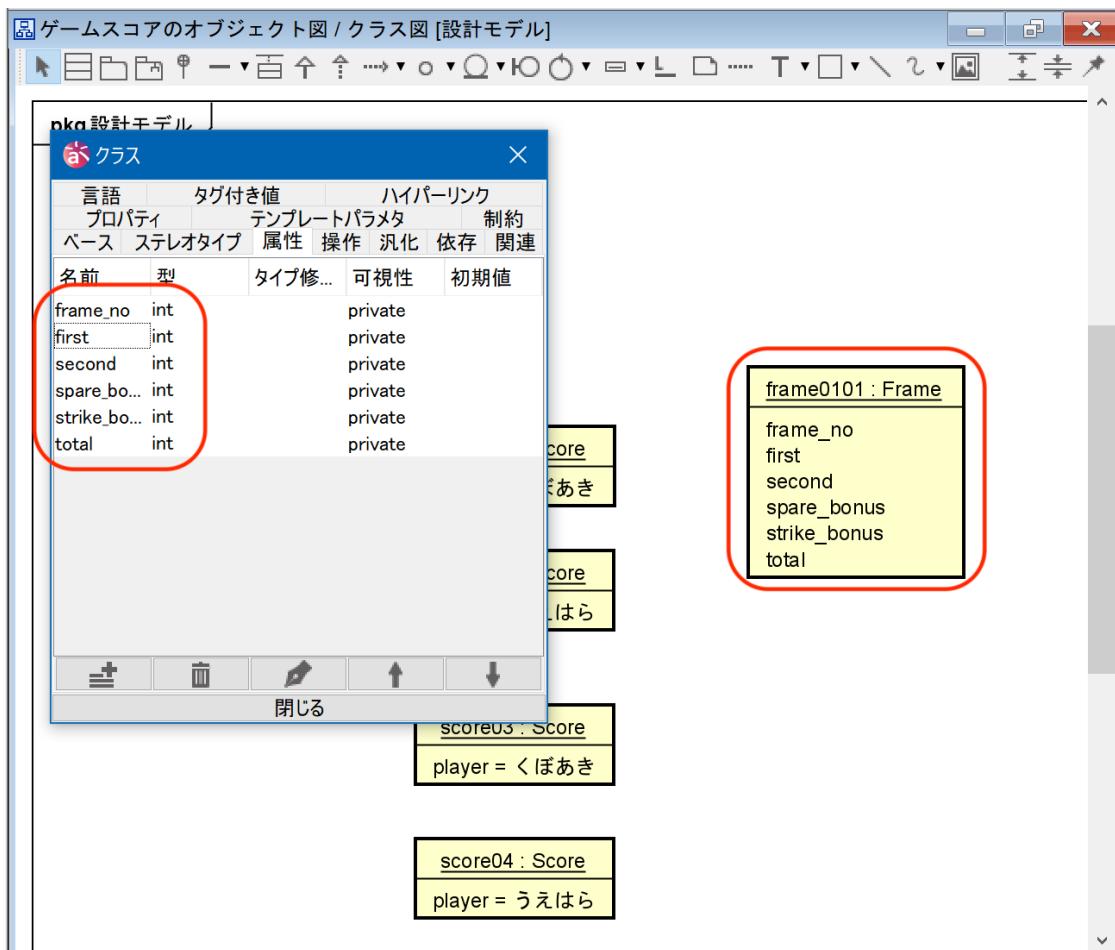


図 2.21 フレームを表すオブジェクト「frame0101」を追加した

追加したオブジェクトのスロットに値を設定する

オブジェクト「frame0101」の各スロットに値を設定します。

- 1組目のゲームの「くぼあき」さんの第1フレームは、1投目7ピン、2投目ミス(0ピン)。
- 第1フレームのトータルは7ピン。ボーナスはなし(2.2.7.2)。

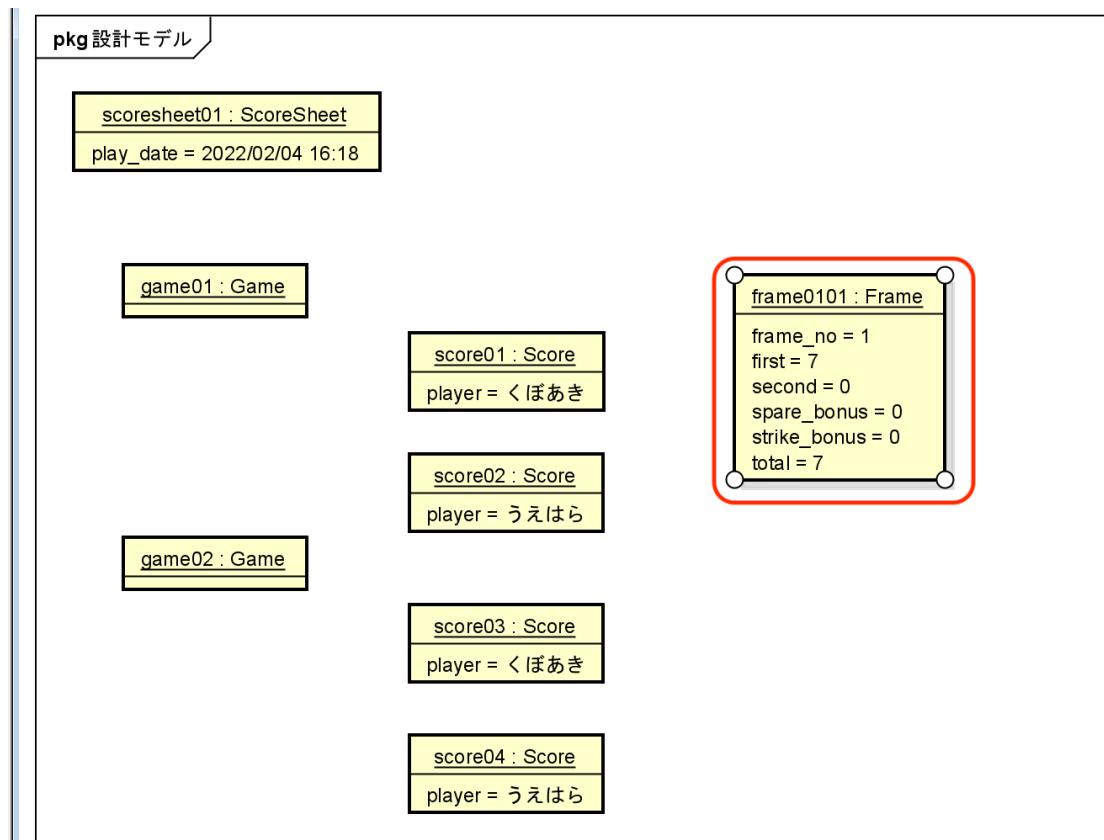


図 2.22 オブジェクト「frame0101」の各スロットの値を設定した

- ・ 同様にして、ほかのフレームのオブジェクトも作成する(図 2.22)。

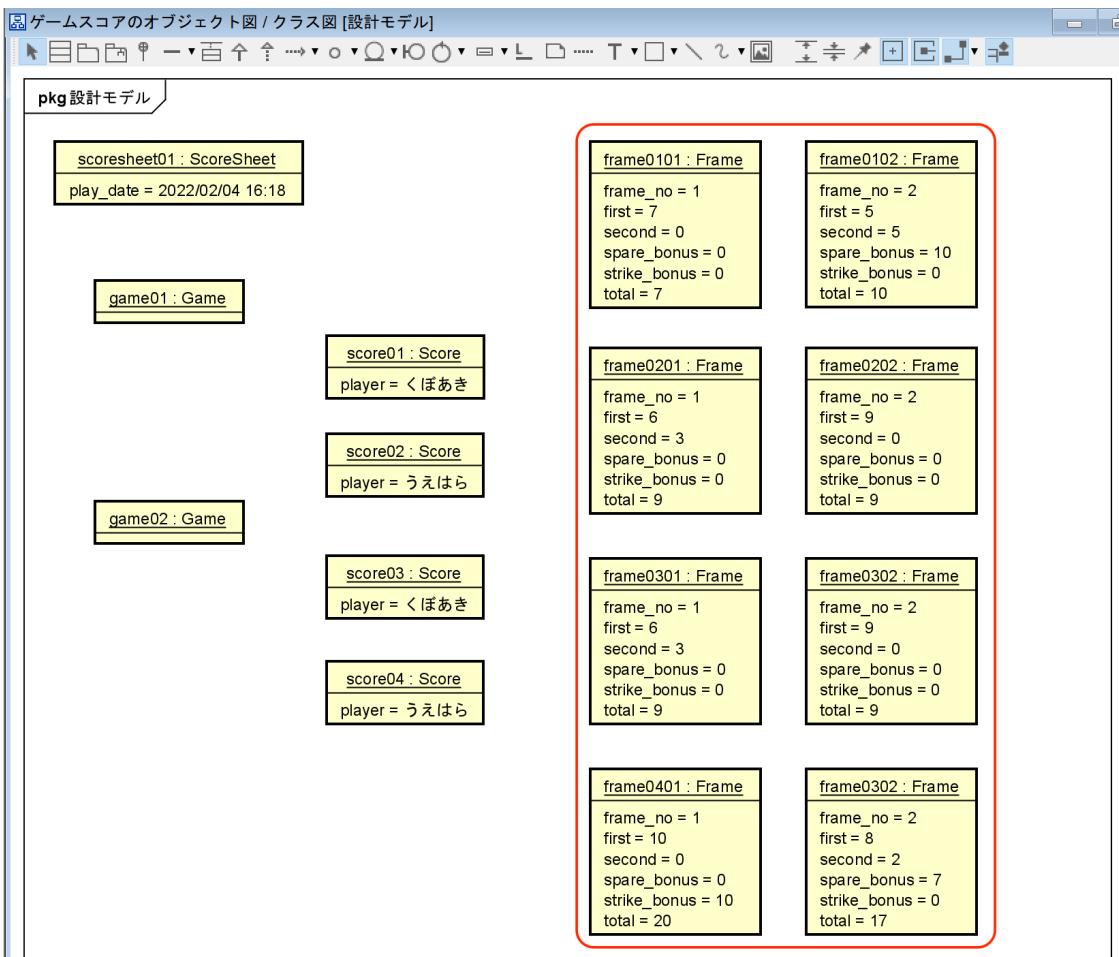


図 2.23 残りのフレームのオブジェクトを作成した(一部だけ)

2.2.8 オブジェクトのつながりを整理する

オブジェクト図に、スコアシート、ゲーム、スコアのオブジェクトが追加できました。図 2.1 を見ながら、これらの間にはどのようなつながりがあるか考えてみましょう。

オブジェクト同士のつながりを考える

- スコアシートは複数のゲームを記録できるので、スコアシートとゲームにはつながりがありそうです。
- ゲームでは、複数のプレーヤーのスコアを記録するので、ゲームとスコアにはつながりがありそうです。
- スコアにはフレームごとのピン数などを記録するので、スコアとフレームにはつながりがありそうです。

つながりがありそうなオブジェクトをリンクを引いてつないでみましょう。

オブジェクト図では、オブジェクト間のつながりを表すには、「リンク」と呼ぶ線でつないで表します。これは、クラス図におけるクラス間のつながりを表す「関連」のインスタンスにあたるものです(クラス図

は、オブジェクト図を元に作成します)。

まず、スコアシートとゲームの間にリンクを引きましょう。

作成したオブジェクトの間にリンクを引く

まず、「scoresheet01」から「game01」へリンクを引いてみましょう。

- ・パレットから「リンク」を選択する。
- ・「scoresheet01」の内部へマウスカーソルを移動し、青枠が表示されるのを待つ(図 2.24)

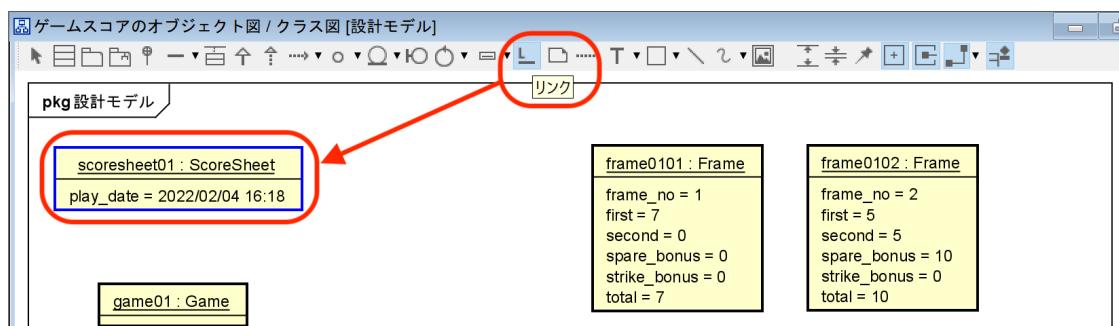


図 2.24 リンクの引き始めのオブジェクトで青枠を表示させる

- ・青枠が表示されたら、マウスのボタンを押したまま「game01」の内部へマウスカーソルを移動する(図 2.25)。

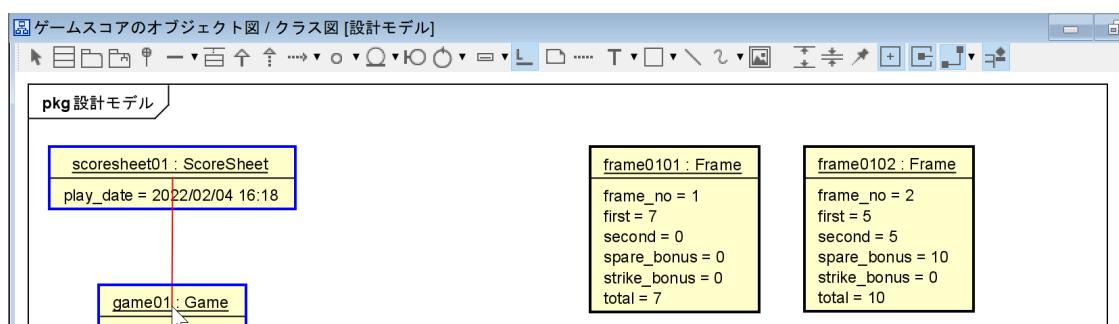


図 2.25 青枠が表示されたらマウスのボタンを押したままマウスカーソルをドラッグする

- ・「game01」にも青枠が表示されたら、マウスのボタンを離すとリンクが引かれる(図 2.26)。

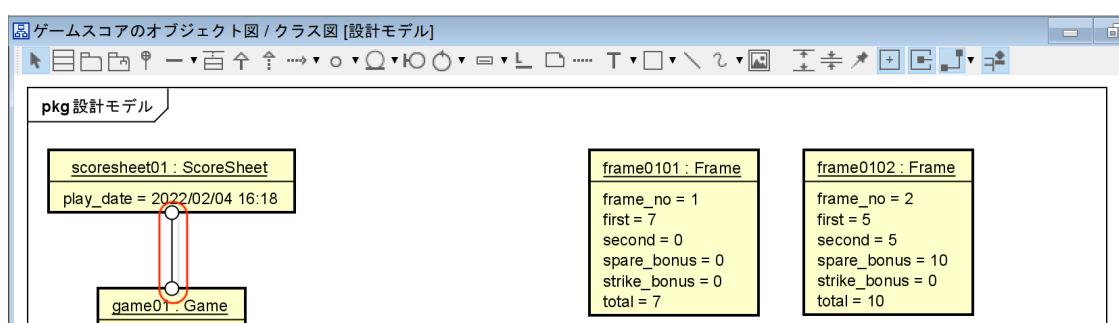


図 2.26 リンク先のオブジェクトにも青枠が表示されたらマウスのボタンを離す

ほかのオブジェクトの間にもリンクを引く

「game02」へも 2.2.8.1 と同様の手順でリンクを引きましょう(図 2.27)。

- ・ 1ゲーム目のスコアは、1つ目のゲームにリンクを引く。
- ・ 2ゲーム目のスコアは、2つ目のゲームにリンクを引く。

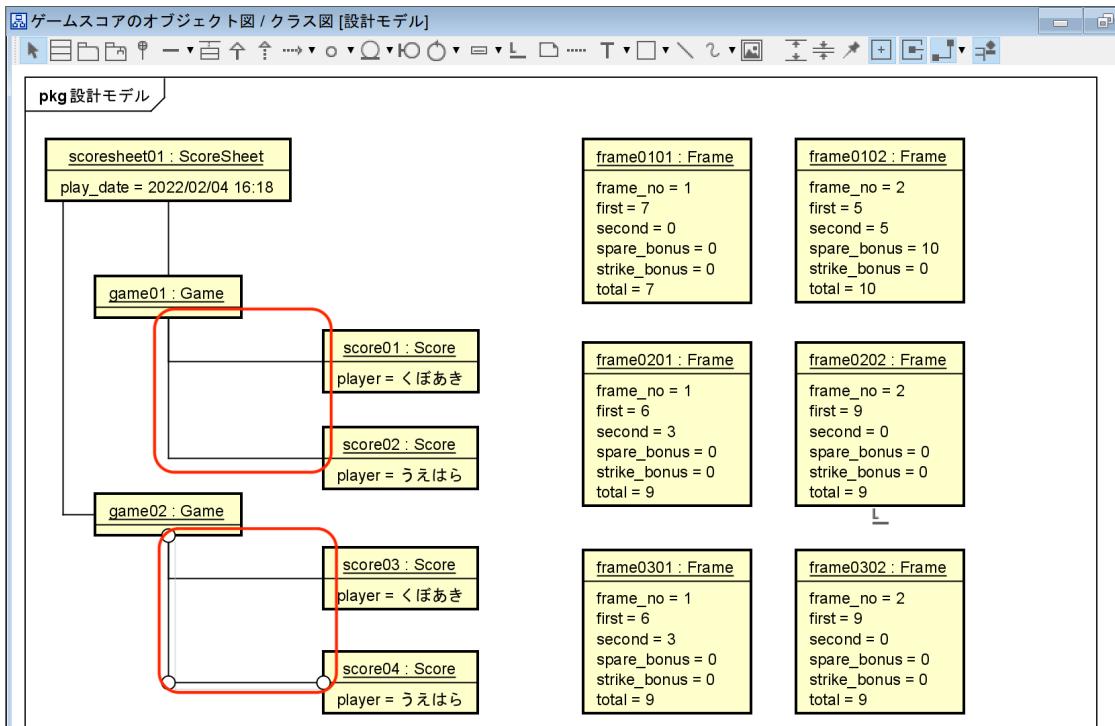


図 2.27 ゲームからスコアへリンクを引く

- ・ スコアからフレームへもリンクを引きます(図 2.28)。

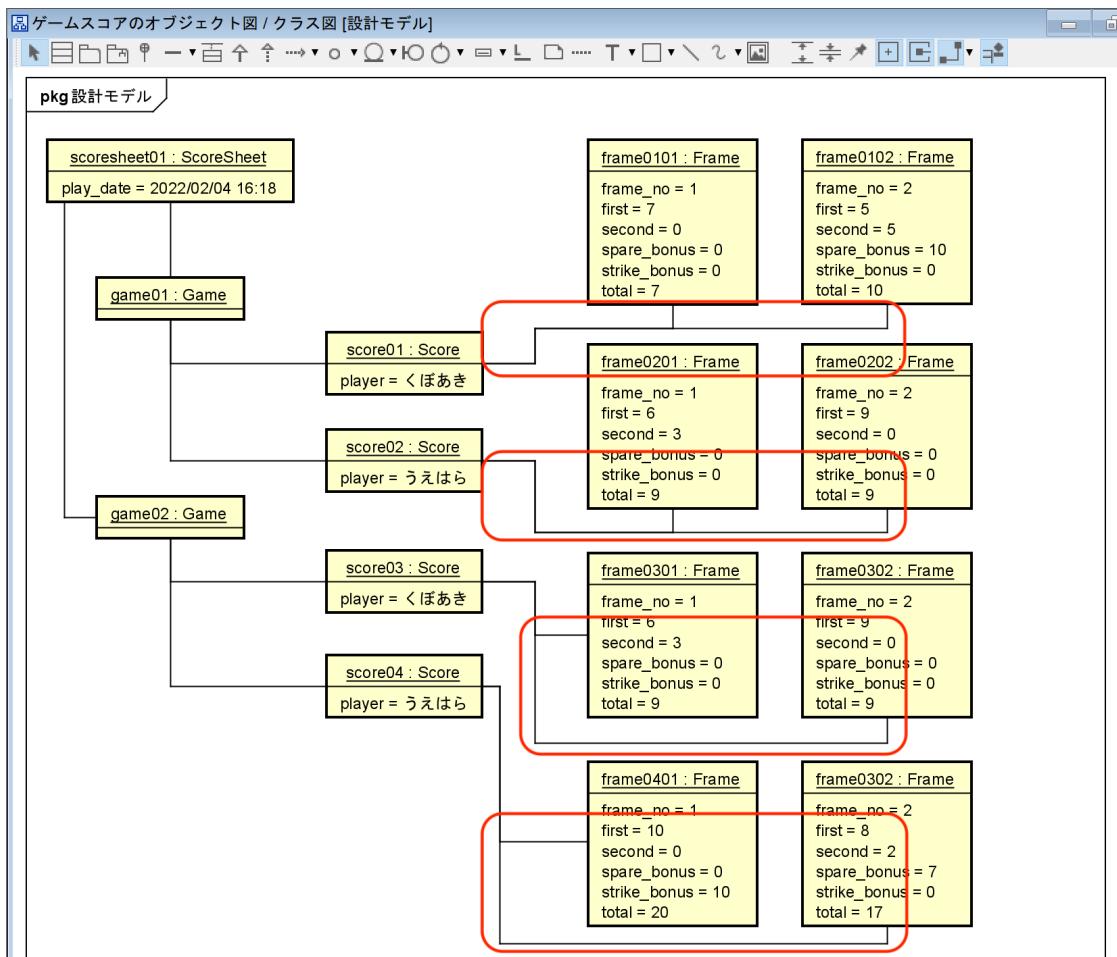


図 2.28 残りのつながりについてリンクを引く

これで、図 2.1 の要素を反映したオブジェクト図が作成できました。

2.3 スコアシートの構造をクラス図で表す

「2.2」では、実際のスコアシートがどうなっているかを調べるために、スコアシートに記載された内容を使ってオブジェクト図を作成しました。しかし、特定のスコアを記載した図では、プログラムで扱うような、より一般的なスコアシートを扱えません。代わりに、オブジェクト図の作成時に考えたクラスを使って、クラス図を使います。そこで、作成したオブジェクト図 図 2.28 を元に、ゲームスコアのクラス図を作成してみましょう。

2.3.1 クラス図とスコアシートクラスを追加する

設計モデルに、「ゲームスコアのクラス図」と「スコアシートクラス」を追加しましょう。

スコアシートのクラス図を追加する

まず、「ゲームスコアのクラス図」を追加します。

- 構造ツリーで、「設計モデル」でポップアップメニューを開き、「図の追加>クラス図」でモデルにクラス図を追加する(図 2.29)。

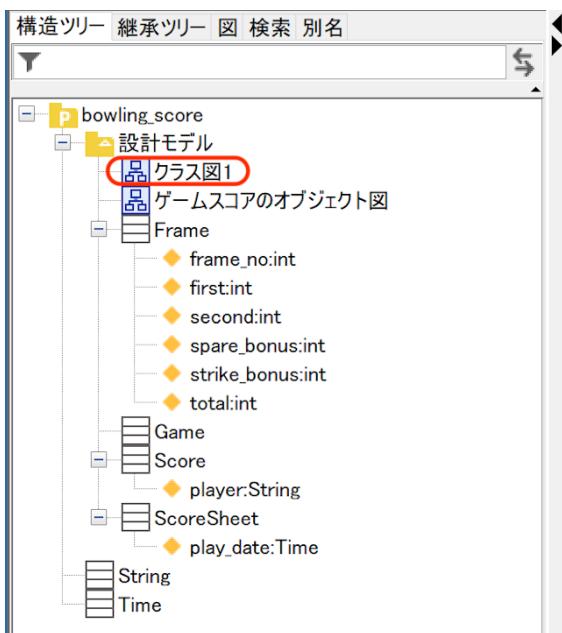


図 2.29 モデルにクラス図を追加する

- 追加した図を「ゲームスコアのクラス図」とする(図 2.30)。

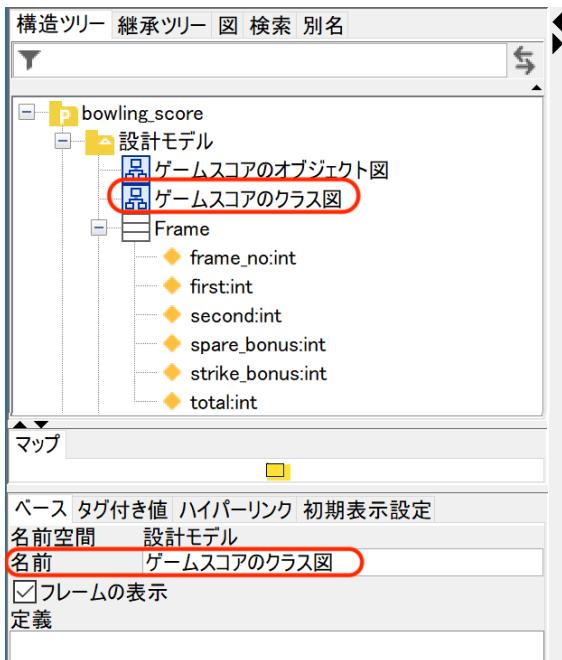


図 2.30 追加した図を「ゲームスコアのクラス図」とした

クラス図にクラスを追加する

追加したクラス図に、ゲームスコアクラスを追加します。

- 構造ツリーから、オブジェクト図を作成したとき登録した「ScoreSheet」クラスをさがし、選択する。
- 選択したクラスをドラッグ&ドロップして、クラス図に追加する(図 2.31)。

- ・構造ツリーから、「Game」クラスを選択する。
- ・選択したクラスをドラッグ&ドロップして、クラス図に追加する(図 2.31)。

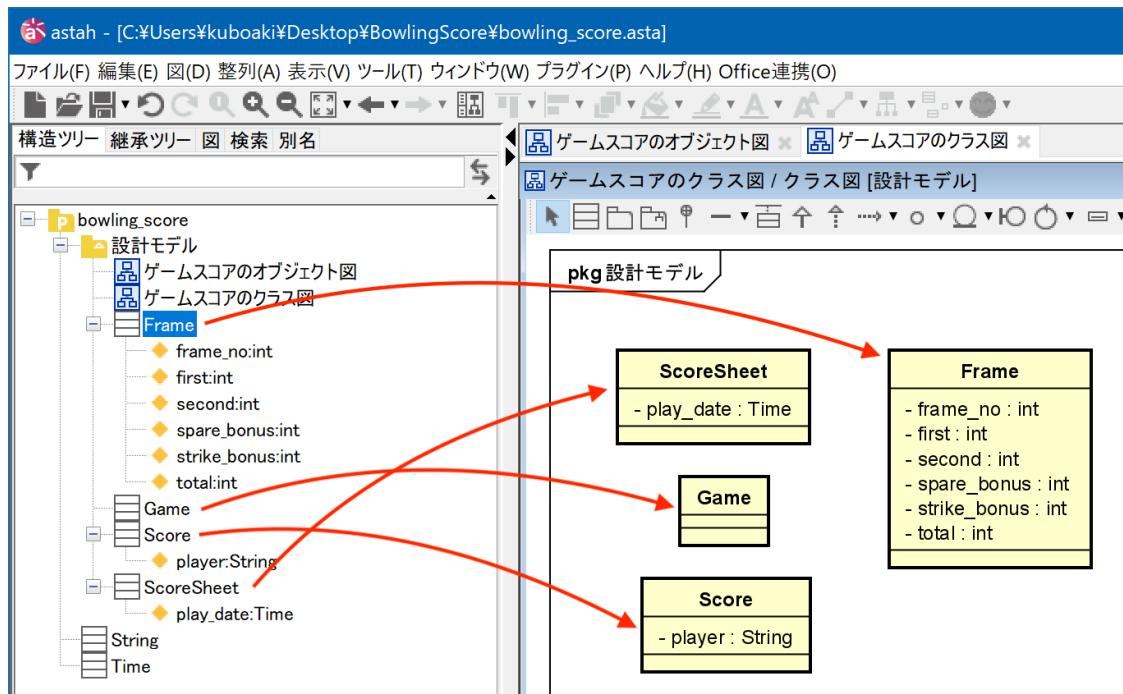


図 2.31 既存のクラスを構造ツリーからクラス図に追加する

2.3.2 クラス間の関連を追加する(1)

作成したオブジェクト図(図 2.27)に記載されているリンクを参照して、リンクでつながっているオブジェクトが属するクラスの間に関連を引きます。

スコアシートとゲームの間に関連を追加する

まず、「ScoreSheet」クラスから「Game」クラスへ関連を引きます。

- ・パレットから矢印付きの関連を選択する。
- ・「ScoreSheet」クラスから「Game」クラスへ向かって関連を引く(図 2.32)。

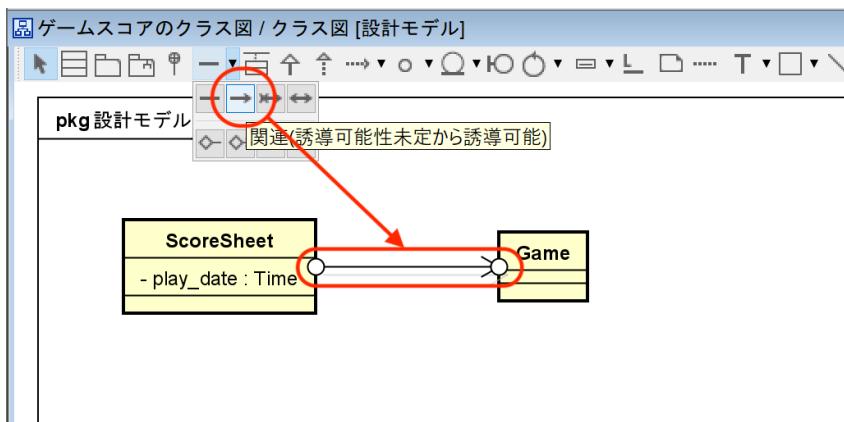


図 2.32 「ScoreSheet」クラスから「Game」クラスへ関連を引いた

スコアシートからみたゲームの多重度を設定する

スコアシートには1ゲーム以上の複数のゲームを記録できます。クラス間の関連において、このことを表には多重度を使います。

- 図 2.32 で引いた関連を選択した状態で、プロパティからターゲットが「Game」の関連端のタブを開く。
- 「多重度」を「1..*」に設定する(図 2.33)。



図 2.33 「ScoreSheet」からみた「Game」の多重度を「1..*」に設定した

スコアシートからみたゲームの関連端名を設定する

スコアシートがゲームを参照するときに使う名前を決めるために、関連端名を設定します。

- 図 2.32 で引いた関連を選択した状態で、プロパティからターゲットが「Game」の関連端のタブを開く。
- ゲームを複数回記録できることを反映して、「名前」を「games」に設定する(図 2.34)。

ベース	ステレオタイプ	制約	関連端 A
関連端 A 制約	関連端 B	関連端 B 制約	タグ付き値
ターゲット	Game		▼
タイプ修飾子			
名前	games		
誘導可能	navigable		▼
集約	none		▼
初期値			
可視性	private		▼
Static	false		▼
Leaf	false		▼
多密度	1..*		▼
派生	false		▼
定義			

図 2.34 「ScoreSheet」からみた「Game」の関連端名を「games」に設定した

スコアシートとゲームの集約関係を設定する

スコアシートには、複数のゲームを追加できます。このことをクラス図に反映する方法がほしいところです。

まず、スコアシートが複数のゲームをとりまとめていること表すのに、「スコアシートには複数のゲームを集約する」役割があるというように捉えてみましょう。このような役割を表すために、クラスの関連端には「集約」を表すための記法が用意されています。

さらに、同じスコアシートで追加でゲームをやりたいときは、新しいゲームを始めた時点で新しいゲームの記録を始めます。つまり、スコアシートが作成される時点と、あるゲームが開始される時点が同時とは限らないわけです。このように、集約する側(ここではスコアシート)のインスタンスと、集約される側(ここではゲーム)のインスタンスが作成される時点が同じではないことを「スコアシートとゲームはライフサイクルが異なる」といいます。一方で、集約する側とされる側が同時に作成され、同時に破棄されるなら、「ライフサイクルが同じ」と考えます。集約の記法には、ライフサイクルが異なる場合の「aggregate」、ライフサイクルが同じ場合の「composit」を表すオプションが用意されています。

ここでは、スコアシートが複数のゲームをとりまとめていて、スコアシートとゲームのライフサイクルが異なることを反映して、スコアシート側の関連端に「aggregate」オプションを指定した集約を設定しよう。

- ・ 図 2.32 で引いた関連を選択した状態で、プロパティ一からターゲットが「ScoreSheet」の関連端のタブを開く(描画時の順序によって該当する関連端がAの場合とBの場合がある)。
- ・ 「集約」のプルダウンメニューから「aggregate」を選択する(図 2.35)。



図 2.35 「ScoreSheet」が「Game」を集約していることを示した

- ・「ScoreSheet」から「Game」への関連が引けた(図 2.36)。

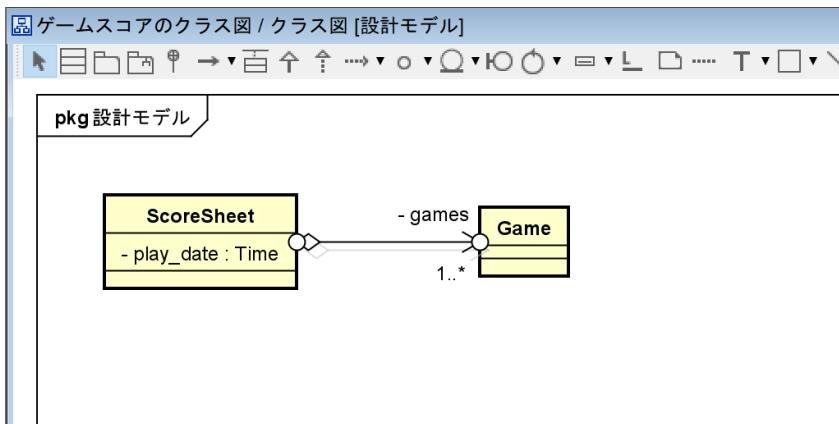


図 2.36 「ScoreSheet」から「Game」への関連が引けた

2.3.3 クラス間の関連を追加する(2)

ゲームとスコア、スコアとフレームの間にも関連を引いて、関連端も設定してみましょう。

ゲームとスコアの間に関連を追加する

こんどは、「Game」クラスから「Score」クラスへ向かって関連を引きましょう。1組のゲームには、複数プレーヤーのスコアが記録できます。つまり、ここにも、スコアシートとゲームの間と同じような関連が引けそうですね。

* 「Score」クラスと「Game」クラスの間に関連を引く。
 * 「Score」クラス側の関連の多重度を「1..*」に設定する。
 * 「Score」クラス側の関連端には関連端名として「scores」を設定する(複数のスコアが記録できることを反映した)。
 * 「Game」クラス側の「集約」のプルダウンメニューから「aggregate」を選択する(図 2.37)。



図 2.37 ゲームとスコアの間に関連を引いた

スコアとフレームの間に関連を追加する

最後に、「Score」クラスから「Frame」クラスへ向かって関連を引きましょう。フレームのオブジェクトをいつ用意するのかには、いくつか選択肢があります。たとえば、ゲームを用意したときに全フレーム分用意すると決めることもできるでしょう。あるいは、ゲームの進行に合わせて、その都度フレームを追加すると決めることもできるでしょう。いくつかの選択肢があるとき、それらからいずれを選択するのかを決定するのは、多くの場合設計時の決定事項です(このこと設計事項、設計項目、設計事由などと呼ぶ人たちもいます)。

このチュートリアルでは、スコアのオブジェクトを用意したときは、常に10フレーム分のフレームのオブジェクトも一緒に用意することとします。この考えを選択したのは、表示や出力の際に、未投球のフレームについても、それらのフレームのオブジェクトを参照すれば、未投球の処理で特別な扱いが減られると考えたからです。この場合、フレームとスコアは同時に作成されるので(ライフサイクルが同じなので)、集約の設定もコンポジションにしておきます。

- ・ 「Game」クラスと「Frame」クラスの間に関連を引く。
- ・ 「Frame」クラス側の関連端の多密度を「10」に設定する。
- ・ 「Frame」クラス側の関連端には関連端名として「frames」を設定する(複数のフレームが記録できることを意識した名前にした)。
 - 集約のプルダウンメニューの中から「composite」を選択する(図 2.38)。
 - 「Score」クラス側の関連端の表示が、黒塗りのダイアモンド(コンポジションのシンボル)に変わる。

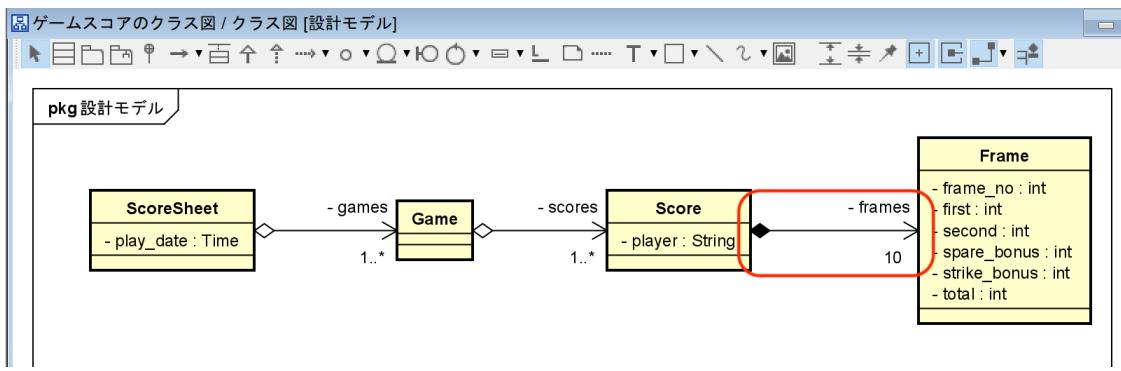


図 2.38 スコアとフレームの間に関連を引いた

これで、いったん、ボウリングのゲームスコアを表した構造のモデルができました。

2.4 まとめ

ボウリングのゲームスコアを記録するスコアシートの構造を検討しました。

2.4.1 構造のモデルを作成した手順

構造のモデルに登場する構成要素や要素間の関連を見つけ出すために、スコアシートの構造のモデルを作成した手順 のような手順を使いました。

スコアシートの構造のモデルを作成した手順

1. スコアシートを観察して、どのような要素で構成されているか洗い出した。
2. スコアシートの実例をそのまま使ってオブジェクト図で表した。
3. オブジェクト図の要素や要素間のつながりを観察して、クラス図を作成した。

まず、オブジェクト図をつくることで、オブジェクトを洗い出し、オブジェクト同士のつながり(リンク)を見つめました。そして、オブジェクト図を参照しながら、クラス図を作成しました。クラス間の関連の関連端名や多重度を検討する際は、オブジェクト図におけるリンクの数などを判断材料にしました。

2.4.2 この段階の構造のモデルはまだ未完成

ここで作成したクラス図には、検討の余地が残っています。たとえば、いずれのクラスにも操作が定義できていません。これは、プレーするのに従ってスコアをつけるときの動作(振る舞い)を検討していないからです。また、動作を検討する中で、属性や関連についても追加や修正が必要になる場合があります。

そこで、続いて振る舞いのモデルを検討します。振る舞いのモデルの作成が進むにつれて、構造のモデルも追加・修正されることになるでしょう。

3 モデルとコードの対応づけ

プログラムがどのように動作するのか検討するときには、振る舞いのモデルを使います。振る舞いのモデルを作成する前に、構造のモデルと振る舞いのモデルがどのようなコードとして実現されるのかについて、このチュートリアルで用いる方式を決めておきましょう。このことを「モデルとコードの対応づけ」と呼ぶことにします。



このような対応づけを「モデル変換ルール」と呼ぶこともあります。また、開発プロセスの中でこの対応づけを検討する工程を「方式設計」と呼ぶ人もいます。

モデルとコードの対応づけを決めてそれを前提にして設計する（モデルを作る）ことは、実装に依存する情報と依存しない情報の分離を促します。また、このような考え方は、実装に依存しないモデルを作る上でとても重要で、実務的なモデルを作成する際にも大いに役立ちます。

3.1 対応づけ検討用モデルの作成

このチュートリアルでは実装にRubyを使うことにしました。サンプルモデルとサンプルモデルに対応するRubyプログラムを作ってみることで、対応づけの方式の基本的な考え方を決めておこうというわけです。

この章で検討したいことは、ボウリングのスコアに関するモデルの問題領域とは直接の関係はありません。こういった別の領域の問題を、これまで検討していたボウリングのスコアに関する問題領域と、ここで検討したいことが混在してしまい、説明も受け取り手も混乱してしまうでしょう。そんなわけで、別の問題として分けて検討するために、この章を用意しています。



もし、この章の役割がピンとこない、実装についての議論にはあまり関心がないのであれば、最初はこの章を読み飛ばしてみてもかまいません。しかし、後の章へ進むと、実装する人がモデルを使う際には、モデルとコードを対応づける必要があり、ここで実装方式を決めておかなければ、実装者がそのことについて考える必要がある（特定の実装者に依存しないよう設計しておくべきことを実装する各人に委ねてしまっている）ことに気づくでしょう。そのときは、この章に戻って、どうして実装方式の設計が必要なのか考えてもらえたたらと思います。

3.1.1 対応づけ検討用プロジェクトを用意する

「モデルとコードの対応づけ」の検討用に astah* で別のプロジェクトを作成しましょう。

対応づけ検討用プロジェクトを作成する

- astah* で新規プロジェクトを作成する。

- ・作成したプロジェクトをいったん保存する。
- ・モデルファイルの名前は「stm_sample.asta」とする。

検討用プロジェクトにクラス図を追加する

プロジェクトが保存できたら、プロジェクトにクラス図を追加しましょう。

- ・構造ツリーで、プロジェクト名をクリックしてポップアップメニューを開く。
- ・「図の追加>クラス図」でクラス図を追加する。
- ・プロパティーから、クラス図の名前を「クラス図」に変更する。

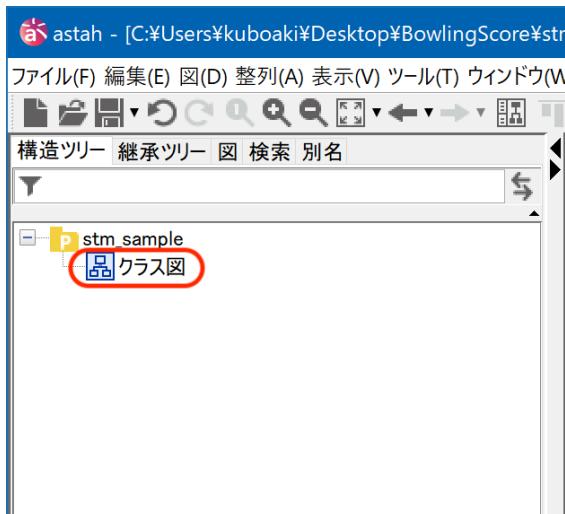


図 3.1 対応づけの説明に使うプロジェクトにクラス図を追加した

クラス図に「Sample」クラスを追加する

対応づけの説明用に、簡単なクラス「Sample」の定義から始めましょう。このクラスは、操作「Play」と属性「attr_a」「attr_b」を持つクラスとします。これをUMLのクラスとして表してみましょう。

- ・パレットからクラスを選択し、クラス図に追加する。
- ・クラス名を「Sample」にする(図 3.2)。
- ・クラスに操作「play」を追加する。
- ・クラスに属性「attr_a」「attr_b」を追加する。

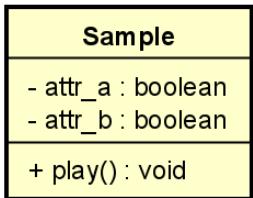


図 3.2 クラス図に「Sample」クラスを追加した

Sampleクラスの最初のサンプルコード

対応づけルールの1つ目として、図 3.2 を表したRubyのコードは リスト 3.1 のように書きたいと決めることにします。

リスト 3.1 【Ruby】「Sample」クラスを表すRubyのコード

```
class Sample ①
  def initialize ②
    @attr_a = true ③
    @attr_b = true ③
  end

  def play ④
  end
end
```

- ① クラス「Sample」の定義の始まり
- ② 初期化メソッド(コンストラクタにあたるメソッド)
- ③ このクラスのインスタンス変数
- ④ メソッド「Play」の定義の始まり

3.1.2 クラスにステートマシン図を追加する

このサンプルにおいては、いま追加した操作「play」が、「Sample」クラスの振る舞いを提供しているとします。しかし、クラス図に描いた「Sample」クラスに操作「play」を追加しただけでは、その処理内容はわかりません。どこかに「play」の処理内容を表した図が必要です。

そこで、「Sample」クラスに対して、操作「play」の振る舞いを表すステートマシン図を追加します。ここで、図を追加する対象を「Sample」クラスにしているのは、このステートマシン図が「Sample」クラスの振る舞いのモデル図であるとわかるようにするためです。

クラスにステートマシン図を追加する

「Sample」クラスにステートマシン図を追加します。

- ・構造ツリーから「Sample」クラスを選択する。
- ・右クリックしてポップアップメニューを表示する(図 3.3)。
- ・「図の追加>ステートマシン図」を選択すると、ステートマシン図が追加される。

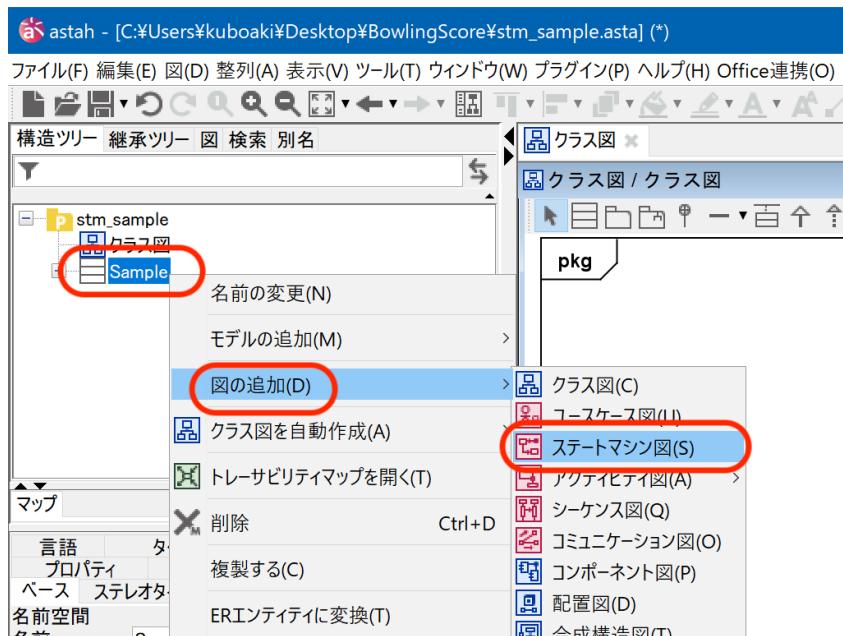


図 3.3 「Sample」クラスにステートマシン図を追加する

- ・プロパティ一から図の名前を編集し、「Sampleのplayのステートマシン図」とする(図 3.4)。
- ・ダイアグラムエディタのタイトルやタブにも図の名前が反映される。

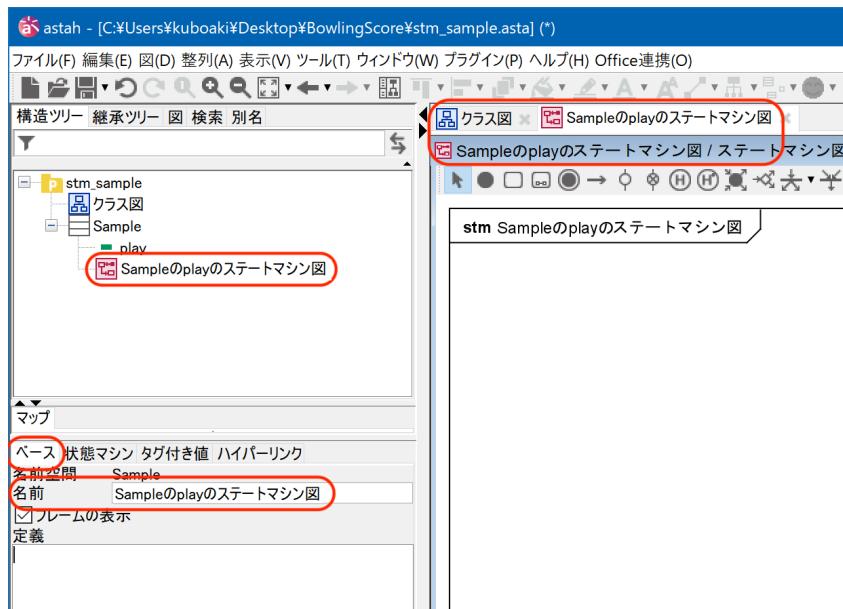


図 3.4 追加したステートマシン図に名前をつける

【参考】ステートマシン図を学ぶチュートリアル

ここでは、モデルとコードの対応づけを説明するために、振る舞いのモデルの一種であるステートマシン図を作成しています。ステートマシン図を使って振る舞いを設計する方法については、次のチュートリアルも参考にしてみてください。

ステートマシン図 & 状態遷移表チュートリアル

<https://www.changevision.co/tutorial-statemachine-japanese.html>



図 3.3 では、Sampleクラスの振舞いを表すステートマシン図とわかるようにと考えて、クラスに対してステートマシン図を割り当てました。この他、astah* では、メソッドに対してステートマシン図を割り当てることもできます。任意のメソッドの処理を示すステートマシン図の割当てとしては、後者のほうが好ましいでしょう。ここでは、クラスの振舞いということを意識して、クラス図に割当ててみています。どちらの方法を使うにしても、クラスの振舞いを担当するメソッドを対応づけることは必要になるでしょう。

3.1.3 ステートマシン図に状態を追加する

次に、追加したステートマシン図に、状態を追加します。

ステートマシン図に状態を追加する

1. パレットから「状態」を選択し、ステートマシン図に状態を追加する。
 - 追加した状態の状態名は「状態0」(数字は追加の都度変わる)となっている(図 3.5)。

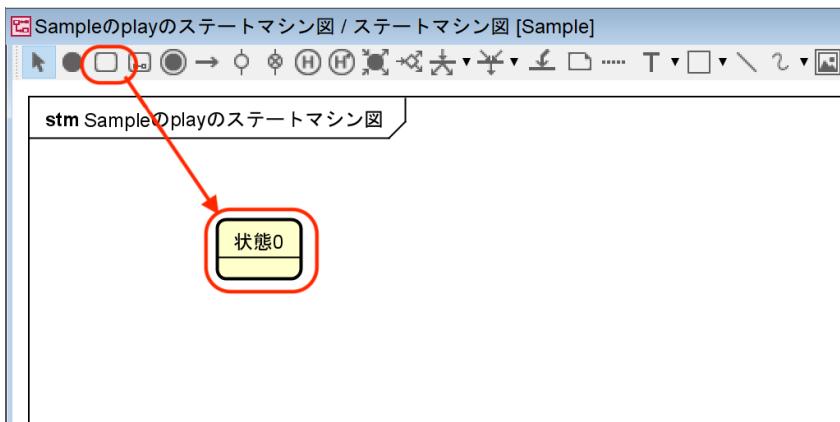


図 3.5 新しい状態を追加した

2. 「状態0」を選択し、プロパティーの「ベース」タブの「名前」を編集して、状態の名前を「ST0」にする(図 3.6)。
 - 状態「ST0」は、このサンプルの例示用に使う状態名の1つ。

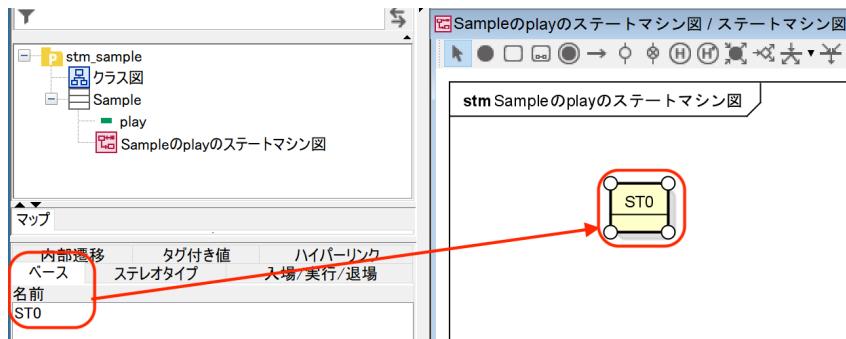


図 3.6 新しい状態に状態名をつけた

3. このサンプルでは3つの状態を使いたいので、状態をもう2つ追加する(図 3.7)。

- 。「ST1」、「ST2」を追加した。

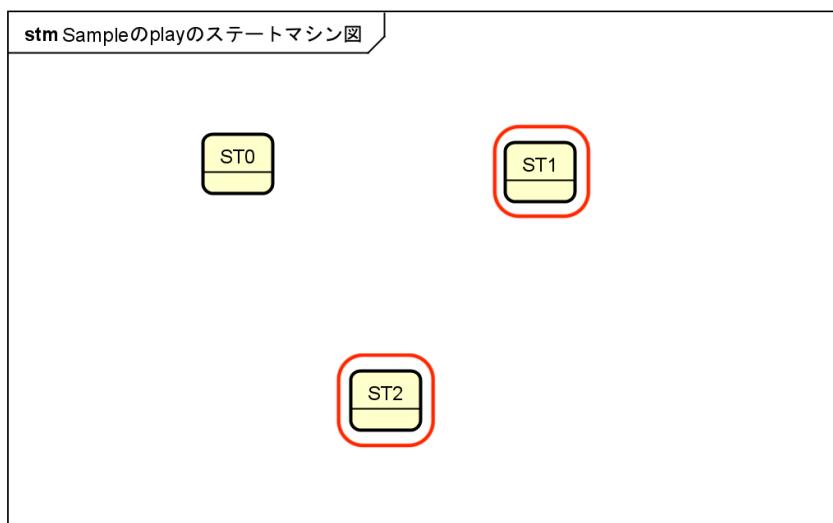


図 3.7 新しい状態に状態名をつけた

3. 開始疑似状態、終了疑似状態を追加した(図 3.8)。

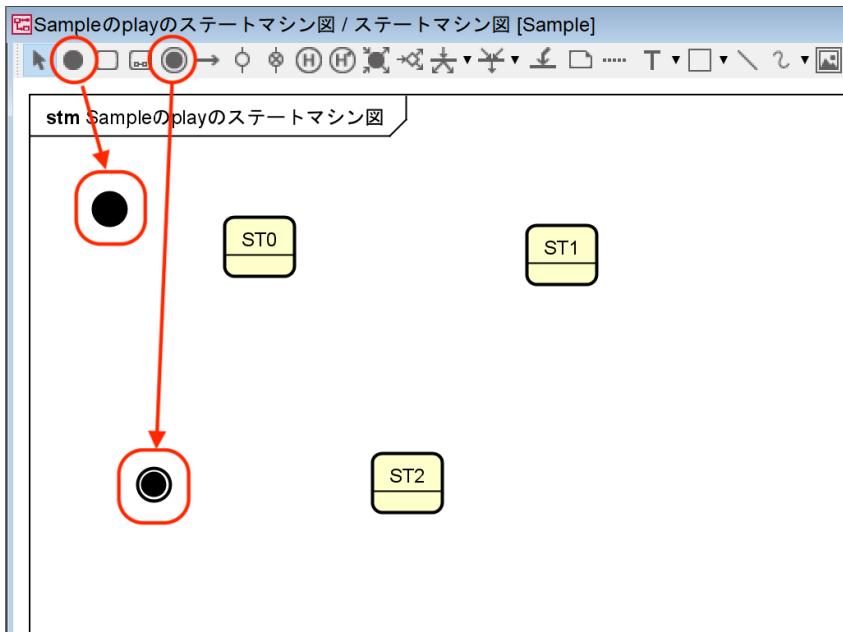


図 3.8 開始疑似状態、終了疑似状態を追加した

3.1.4 ステートマシン図に状態遷移を追加する

状態が追加できたので、こんどは状態遷移を追加しましょう。ステートマシン図には、通常の状態のほかにいくつかの「疑似状態」が登場します。たとえば、「開始疑似状態」は、この図における最初の状態を示すのに使います。「終了疑似状態」は、その図における最後の状態(1つとは限りません)を示すのに使います。

状態遷移を追加する

1. パレットから「遷移」を選択して、「開始疑似状態」の内部へマウスカーソルを移動し、青枠が表示されるのを待つ(図 3.9)。

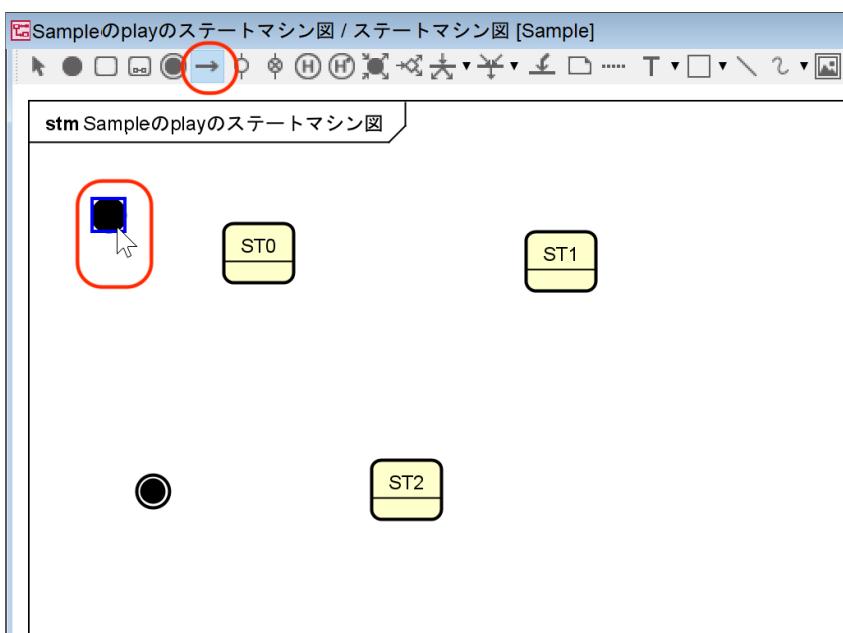


図 3.9 開始疑似状態の内側で青枠を表示させる

2. 青枠が表示されたらマウスカーソルをドラッグする(マウスのボタンを押したままマウスカーソルを移動する)。
3. マウスカーソルを「ST0」へドラッグして青枠が表示されるのを待つ(図 3.10)。

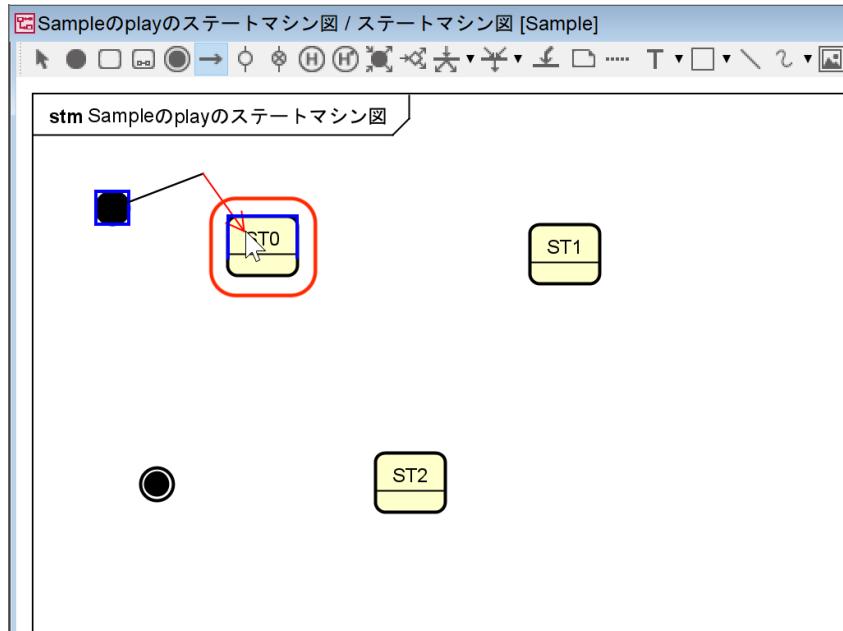


図 3.10 「ST0」へマウスカーソルをドラッグして青枠が表示されるのを待つ

4. 「ST0」でも青枠が表示されたらマウスのボタンを離すと状態遷移が引かれる(図 3.11)。

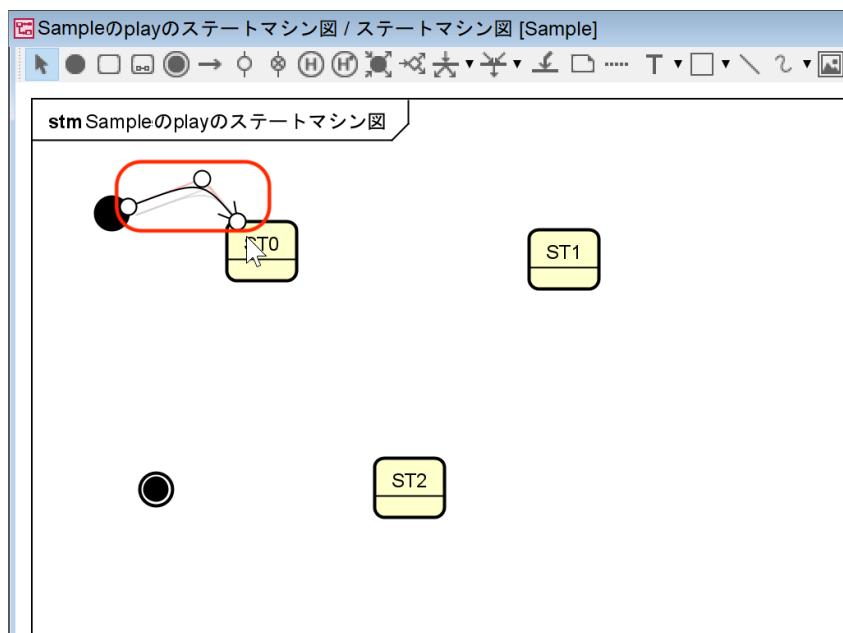


図 3.11 「ST0」でも青枠が表示されたらマウスのボタンを離すと状態遷移が引かれる

4. ほかの状態についても 図 3.12 と同じように状態遷移を追加する。

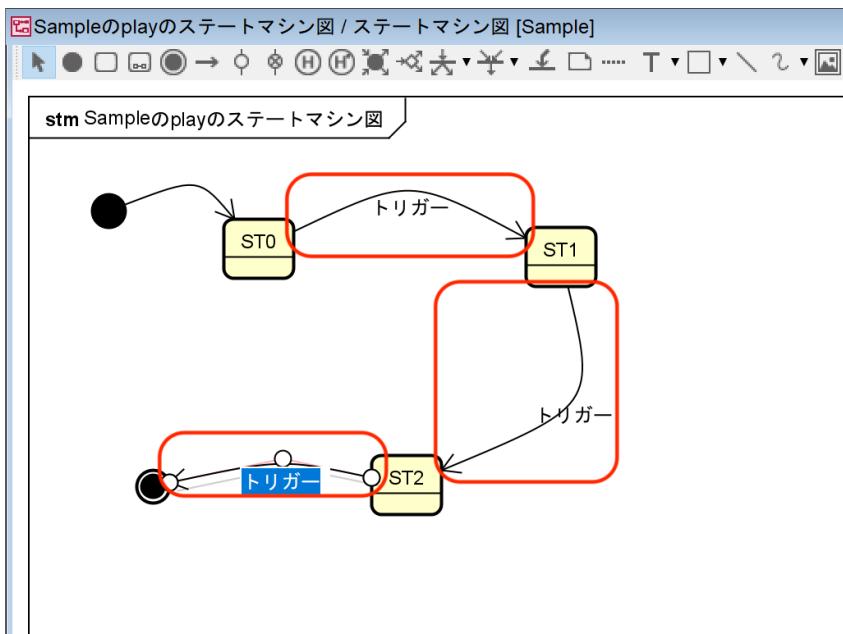


図 3.12 ほかの状態遷移も追加する

3.1.5 状態とイベントと状態遷移の関係

状態とイベント

ステートマシン図において、状態は「イベント」の発生を待っているところです。多くの場合、イベントは、そのクラス自身の操作では処理を先に進められなくなるようなできごとです。たとえば、外部からの入力(操作待ちや受信待ちなど)や、一定の時間経過などがイベントの候補になります。

状態遷移

状態遷移は、ある状態において待っていたイベントが発生して、別の状態へ移ることを表しています。イベントには、イベントが発生したとき、実際に遷移するか判定する条件を追加できます。この条件のことを「ガード条件」と呼びます。ガード条件つきのイベントでは、イベントが発生したときにガード条件を評価し、条件が真なら状態遷移します。ガード条件が偽のとき、イベントは起きたことになります(消費されると呼びます)が、状態は遷移しません。

そして、状態遷移には、イベントが発生したときに実行したい処理を追加できます。この処理のことを「アクション(またはエフェクト)」と呼びます。

イベントが発生すると、ガード条件つきのイベントならさらにガード条件も真なら、状態遷移に伴うアクションが実行され、それから次の状態へ遷移します。

3.1.6 ステートマシン図にイベントやアクションを追加する

まず、「ST0」から「ST1」への状態遷移を、イベントが「ev1」でガード条件は「gd(が真)」のとき、アクション「act1」を実行するよう編集してみましょう。

状態遷移にイベントやアクションを追加する(1)

1. 「ST0」から「ST1」への状態遷移を選択し、この遷移のプロパティーを表示し、ベースタブを開く。
2. プロパティーを編集する(図 3.13)。
 - ・「トリガー」にイベント名を設定する。ここでは「ev1」というイベントを設定する。
 - ・「ガード」にガード条件を設定する。ここでは「gd1(が真)」を設定します。
 - ・「アクション」にアクションを設定する。ここでは「act1」という処理があるとして、これを設定する。

■ act1は、2つの引数(イベントとパラメーター)を持つメソッドと想定する。

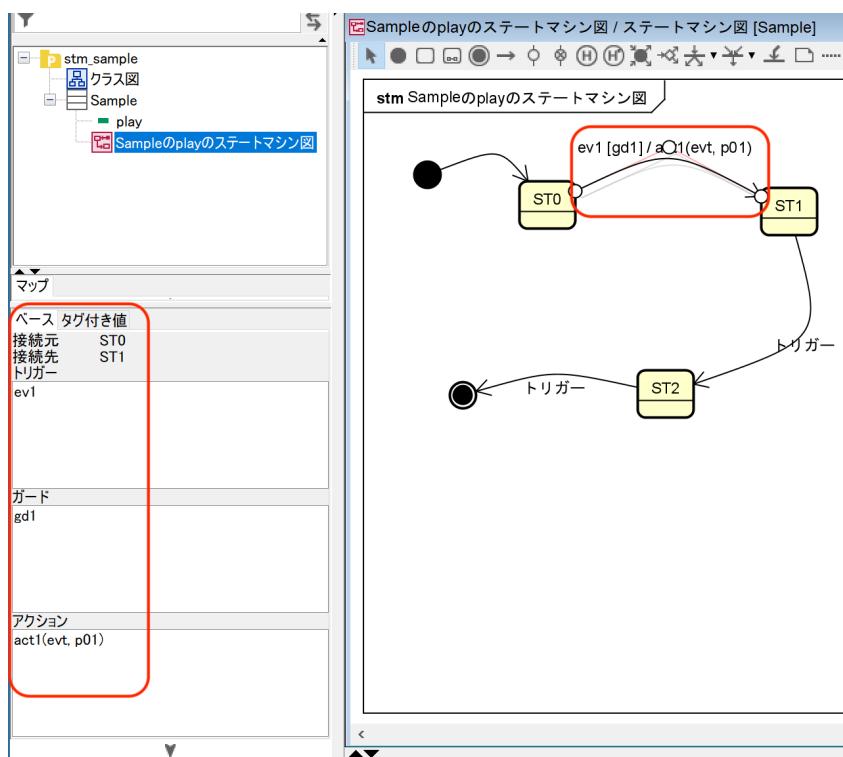


図 3.13 「ST0」から「ST1」への状態遷移のイベントとアクションを編集する

次に、「ST1」からの「ST2」への遷移です。この遷移に割り当てるイベントは、「ev2」、アクションを「act2」とします。そして、このアクションの実行後、ガード条件「gd2」が真なら「ST2」へ、偽ならアクション「act3」を実行してから「ST1」へ遷移するように変更してみます。

遷移先が2つあるなら、状態遷移を別々に引けばよさそうです。ところが、同じイベントを待ってる遷移先が2つ以上あると、どちらの状態へ遷移するのか決定できなくなります(あいまいな状態遷移と呼びます)。そこで、1つのイベントによる遷移先が状況によって変わる場合には「選択疑似状態」を使います。

状態遷移にイベントやアクションを追加する(2)

1. パレットから「選択疑似状態」を選択し、ステートマシン図に追加する(図 3.14)。

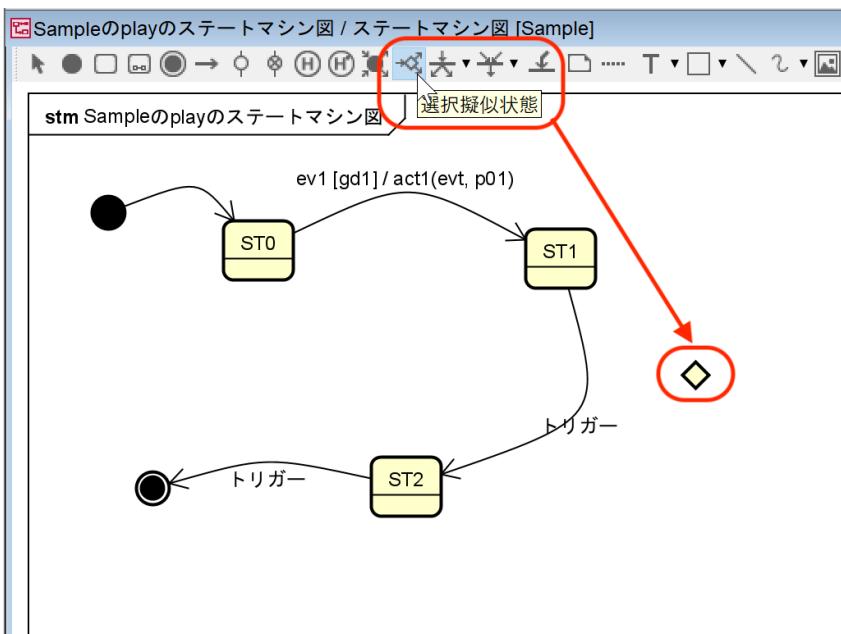


図 3.14 「選択疑似状態」をステートマシン図に追加する

2. 「ST1」から「ST2」への状態遷移を選択し、「ST2」側のハンドル(丸印)をマウスでつまみ、「選択疑似状態」へつなぎ直す(図 3.15)。

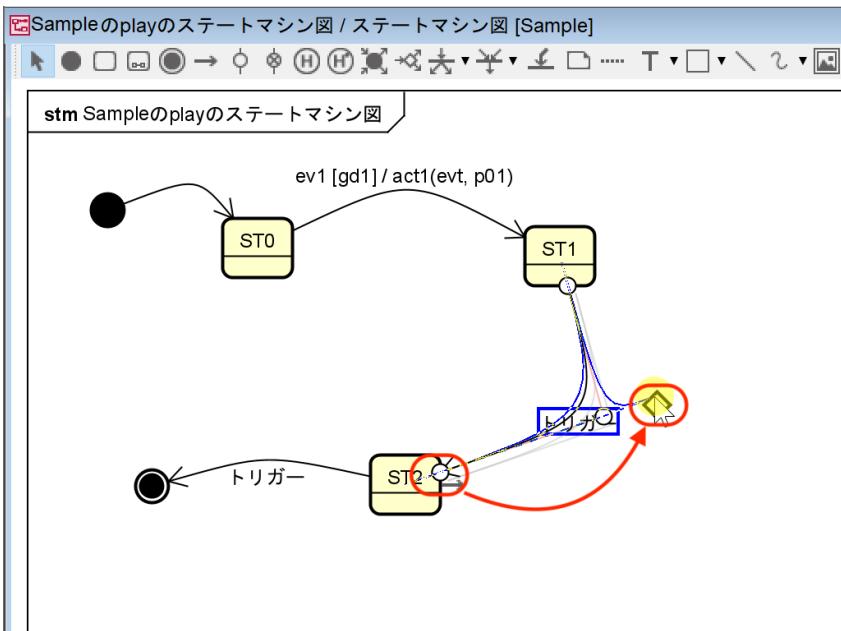


図 3.15 既存の状態遷移を追加した「選択疑似状態」へつなぎ直す

3. 「選択疑似状態」から「ST0」と「ST2」への状態遷移を追加する(図 3.16)。

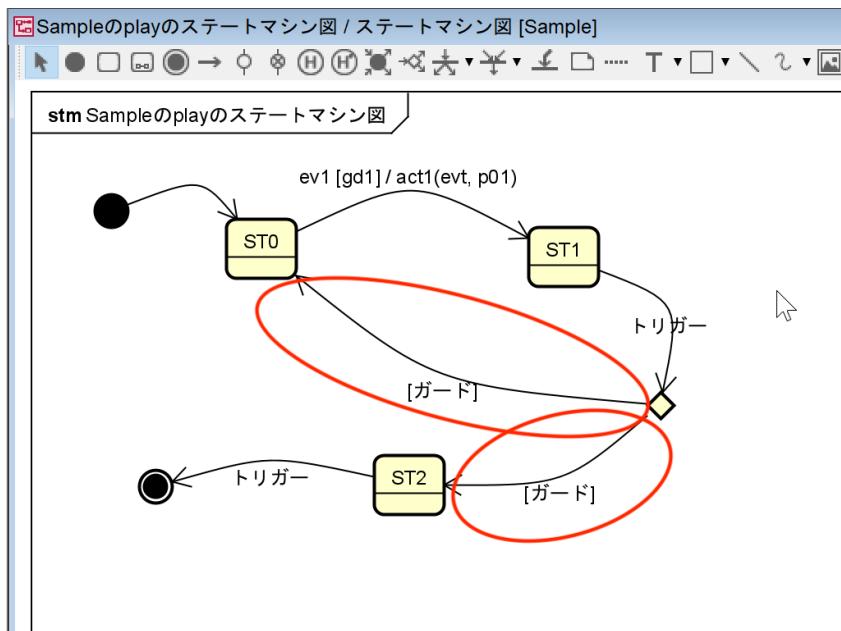


図 3.16 「選択疑似状態」から「ST0」と「ST2」への状態遷移を追加する

4. それぞれの遷移のプロパティを編集する(図 3.17)。

- トリガーに「ev2」、ガード条件はなし、アクションを「act2」に設定する。
- ガードにガード条件「!gd2(gd2ではない)」、アクションを「act3」に設定する。
- ガードにガード条件「gd2」に設定する。

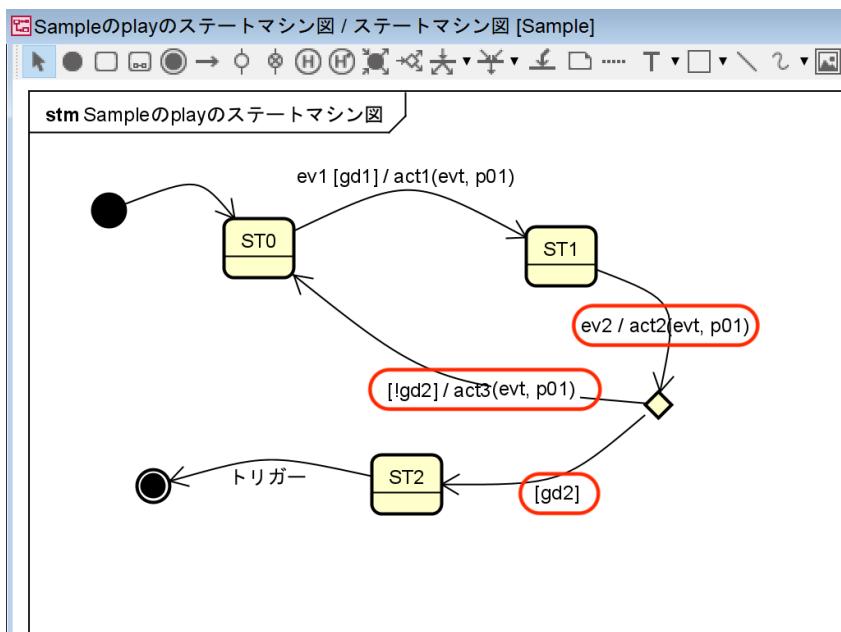


図 3.17 状態遷移のイベントとアクションを編集する

5. 同様にして、もう2つほど状態遷移を追加する(図 3.18)。

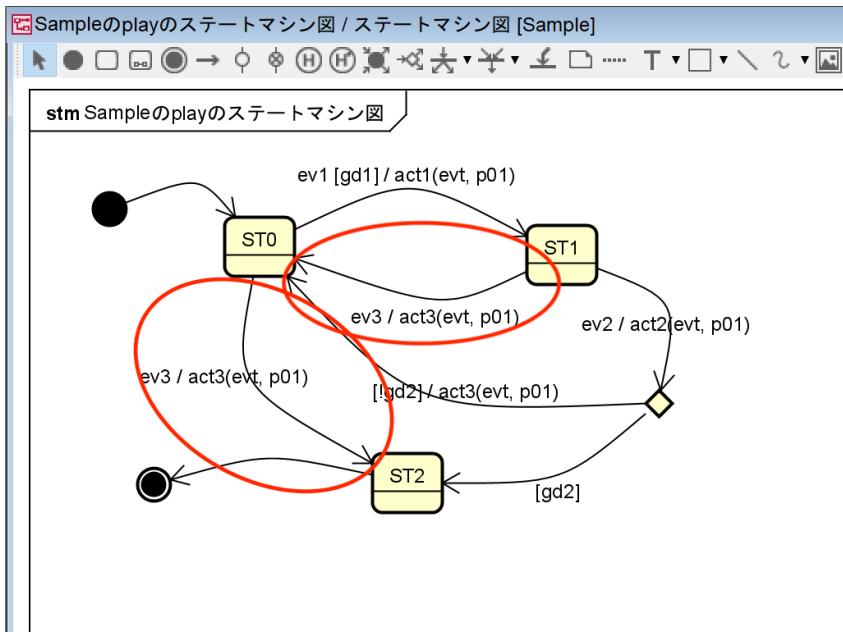


図 3.18 さらに状態遷移とイベントとアクションを追加した

3.1.7 ステートマシン図に対応するコードを作成する

ステートマシン図ができたので、この図に合うようなRubyのコードを作成するルールを考えましょう。

状態を保持する変数を追加する

まず、「Sample」クラスに状態を保持するインスタンス変数を追加します（リスト3.2）。初期値は、最初の状態（ST0）にします。

リスト3.2【Ruby】状態を保持するインスタンス変数を追加する

```

class Sample
  def initialize
    @state = :ST0 ①
    @attr_a = true
    @attr_b = true
  end

  # 略

end
  
```

- ① 現在の状態を保持する変数を用意し、ST0で初期化する。「:ST0」はRubyのシンボルの表記法。シンボルは名前付きの数値定数。

状態遷移を担当するメソッドを作成する

ステートマシン図に書いた状態遷移を担当する操作「play」をplayメソッドとして作成します（リスト3.3）。ルールの検討用なので、途中には処理の確認用の表示処理を書いておきます。

リスト 3.3 【Ruby】状態遷移を担当するメソッドを作成する

```

class Sample
  def initialize
    # 略
  end

  def play(evt, param)
    puts "#{@state} ->" ①
    puts "  event:#{@evt}, param: #{@param}" ②
    case @state ③
    when :ST0
      st0_proc(evt, param) ④
    when :ST1
      st1_proc(evt, param)
    when :ST2
      # none
    end
    puts "      -> #{@state}" ⑤
    puts 'finished.' if @state == :ST2 ⑥
  end
end

```

- ① 遷移前の状態を表示する。
- ② イベントとパラメーターを表示する。
- ③ 状態ごとの処理を case文を使って分岐する。
- ④ 状態ごとの詳細な処理を担当するメソッドを呼び出す(ここでは st0_proc)。
- ⑤ 遷移後の状態を表示する。
- ⑥ 終了状態になったことを知らせる。

ガード条件の判定用メソッドを作成する

このサンプルのステートマシン図には、状態遷移のガード条件として「gd1」「gd2」が登場します。これらをrubyのメソッドとして作成します。rubyでは、真偽値を返すメソッドには「?」をつける習慣がありますので、これにしたがってメソッド名は「gd1?」「gd2?」とします。サンプルとして、属性の値を使った演算を割り当てました(リスト 3.4)。

リスト 3.4 【Ruby】ガード条件のメソッドを作成する

```
class Sample
  def initialize
    # 略
  end

  def play
    # 略
  end

  def gd1? ①
    @attr_a && @attr_b ②
  end

  def gd2? ③
    !@attr_a || @attr_b
  end
end
```

- ① 「gd1」用のメソッド。真偽値を返すメソッドに「?」をつけるRubyの習慣に倣った。
- ② 属性値を使った条件式の例。
- ③ 「gd2」用のメソッド。

状態遷移を追加する(1)

状態ごとの処理を受け持つメソッドを作成します。メソッド名は、状態名を小文字して「_proc」を付けるというルールにします。したがって、状態名「ST0」の場合、メソッド名は「st0_proc」になります(リスト 3.5)。

リスト 3.5 【Ruby】「ST0」の状態遷移のメソッドを作成する

```
class Sample
  def initialize
    # 略
  end

  def play
    # 略
  end

  def st0_proc(evt, param)
    case evt ①
    when :ev1 ②
      if gd1? ③
        puts "    gd1: #{gd1?}"
        act1(evt, param) ④
        @state = :ST1 ⑤
      else ⑥
        puts "    <<< gd1: #{gd1?}, transition is ignored. >>>"
      end
    when :ev3 ⑦
      act3(evt, param)
      @state = :ST2
    end
  end

  # gd1?, gd2? が続く
end
```

- ① イベントで場合分けする。
- ② イベントが「ev1」の場合。イベントはRubyのシンボルを使って表す。
- ③ ガード条件による判定。
- ④ アクション「act1」の呼び出し。
- ⑤ 次の状態「ST1」への遷移。
- ⑥ ガード条件が偽だったとき。イベントが無視されたことを表示する。
- ⑦ イベントが「ev3」の場合。

この部分のRubyのコードとステートマシン図の対応を図で表してみましょう(図 3.19)。

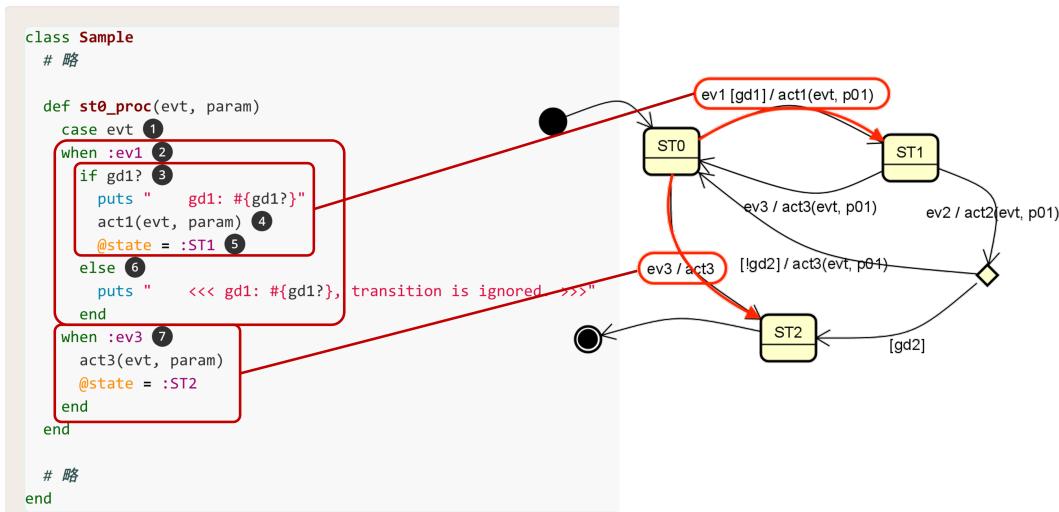


図 3.19 Rubyのコードとステートマシン図の対応関係(1)

おおむねのことは 図 3.19 を読めばわかるでしょうが、対応関係を説明しておきます。

ガード条件のある状態遷移をRubyのコードで表すルール

- 遷移元の状態で待っているイベントごとに場合分けする。
- ガード条件なしのときは(アクションがあればそれを呼び出して)、次の状態へ遷移する。
- ガード条件があるときは、先にガード条件を評価して、真であればアクションを実行して次の状態へ遷移する。
- ガード条件が偽のときは、アクションは実行されず、状態も遷移しない。

状態遷移を追加する(2)

こんどは、「ST1」からの状態遷移に対するメソッド「st1_proc」です(リスト 3.6)。

リスト 3.6 【Ruby】「ST1」の状態遷移のメソッドを作成する

```

class Sample
  def initialize
    # 略
  end

  def play
    # 略
  end

  def st0_proc(evt, param)
    # 略
  end

  def st1_proc(evt, param)
    case evt
    when :ev2 ①
      act2(evt, param) ②
      puts "  gd2: #{gd2?}"
      if gd2? ③
        @state = :ST2 ④
      else
        act3(evt, param) ⑤
        @state = :ST0 ⑥
      end
    when :ev3 ⑦
      act3(evt, param)
      @state = :ST0
    end
  end

  # gd1?, gd2? が続く
end

```

- ① イベントが「ev2」の場合。
- ② アクション「act2」の呼び出し。
- ③ 選択疑似状態におけるガード条件による判定。
- ④ 次の状態「ST0」への遷移。
- ⑤ 選択疑似状態からの遷移におけるアクション「act3」の呼び出し。
- ⑥ 次の状態「ST2」への遷移。
- ⑦ イベントが「ev3」の場合。

この部分のRubyのコードとステートマシン図の対応を図で表してみましょう(図 3.20)。

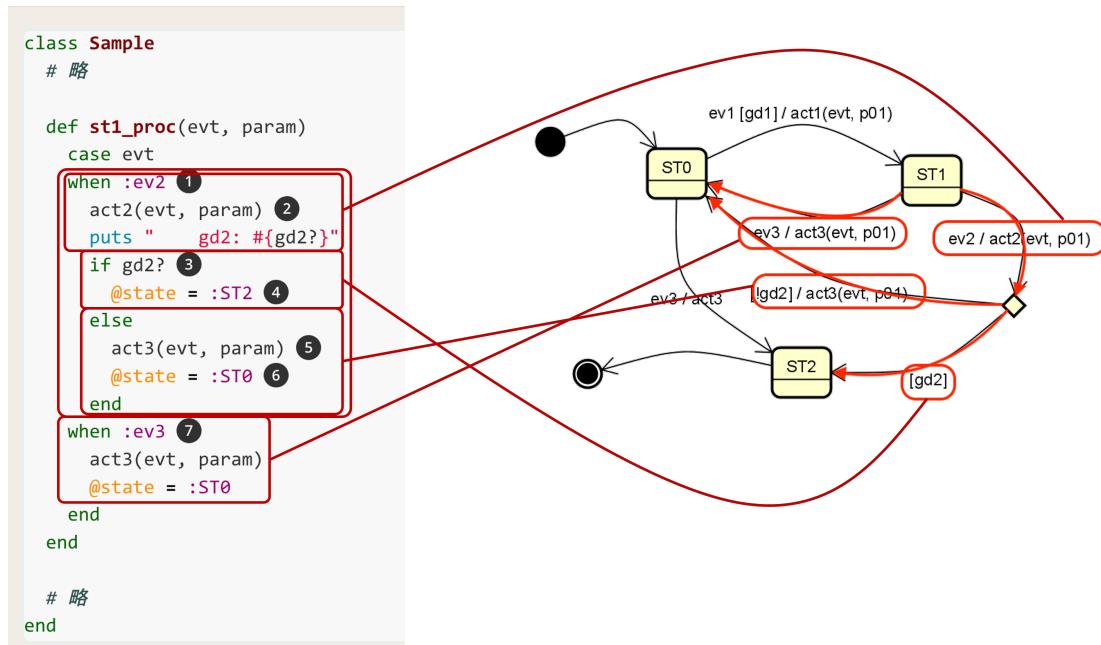


図 3.20 Rubyのコードとステートマシン図の対応関係(2)

おおむねのことは 図 3.20 を読めばわかるでしょうが、対応関係を説明しておきます。

選択疑似状態がある状態遷移をRubyのコードで表すルール

1. 選択疑似状態の前の状態遷移については、図 3.19 のルールで賄う。
2. その後で、選択疑似状態の後の状態遷移のガード条件を評価する。
3. 評価結果によって（アクションがあれば実行してから）、次の状態へ遷移する。
4. 選択疑似状態の後の状態遷移は、ガード条件による場合分けで漏れる場合がないように注意する。

このサンプルでは、アクション「act1」、「act2」、「act3」は、定めたルールを確認しやすいメソッドにしておきます（リスト 3.7）。これらのアクションと、先に挙げたガード条件をチェックするメソッドは、「Sample」クラス内部のメソッドだけが使います。そこで、Rubyのコードでもこれらのメソッドのスコープを「private」にしましょう。Rubyでは、「private」キーワードを追加してその後にメソッドを配置します。

リスト 3.7 【Ruby】「ST1」の状態遷移のメソッドを作成する

```

class Sample
  # initialize、play、st0_proc、st1_proc の定義を省略

  private ①

  def act1(evt, prm)
    puts "act1: event:#{evt}, param: #{prm}"
  end

  def act2(evt, prm)
    puts "act2: event:#{evt}, param: #{prm}"
  end

  def act3(evt, prm)
    puts "act3: event:#{evt}, param: #{prm}"
  end

  def gd1? ②
    @attr_a && @attr_b
  end

  def gd2?
    !@attr_a || @attr_b
  end
end

```

① この宣言以降のメソッドは、可視性が「private」なメソッドになる。

② ガード条件の判定用メソッドも「private」なメソッドに含めた。

これで、ステートマシン図で表したクラスの振る舞いをRubyのコードに対応づけられました。

3.1.8 追加したアクションをクラス図に反映する

ステートマシン図を作成するときに追加したアクションは、このステートマシン図を割り当てているクラス(ここでは「Sample」クラス)の操作にしておきましょう。ここで、「Sample」クラスをテストするクラス「SampleTest」も追加しておきます。

イベントやアクションをクラス図に反映する

1. クラス図を開く。
2. 「Sample」クラスを選択し、プロパティーから「操作」タブを表示する。
3. 「+」アイコンを使って操作「act1」を追加する。
4. 可視性を「private」に設定する。
5. 「act2」、「act3」についても同じように設定する(図 3.21)。

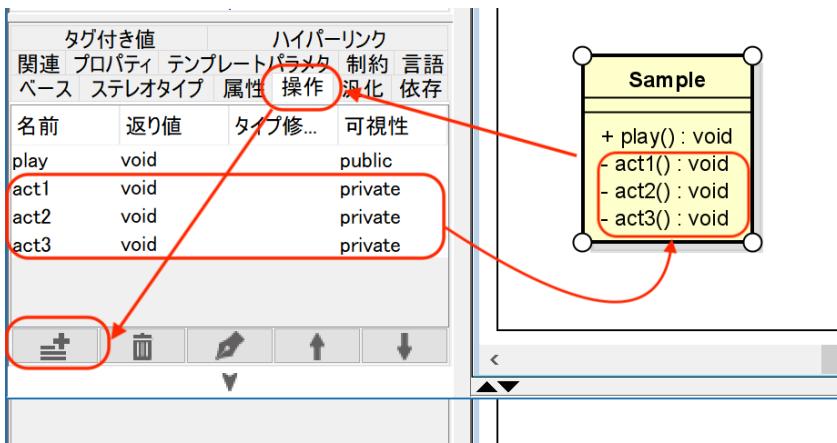


図 3.21 「Sample」クラス内部で使う操作「act1」、「act2」、「act3」を追加する

6. ガード条件に使うメソッド「gd1?」「gd2?」も追加しておく。
7. 「SampleTest」クラスを追加する(図 3.22)。
 - このテスト用クラスには、テストを実行する「run」メソッドを用意しておく。
 - テスト用のメソッドをいくつか追加しておく。
8. 「SampleTest」から「Sample」クラスへ関連を引き、関連端名を「samp01」、多重度を「1」としておく。

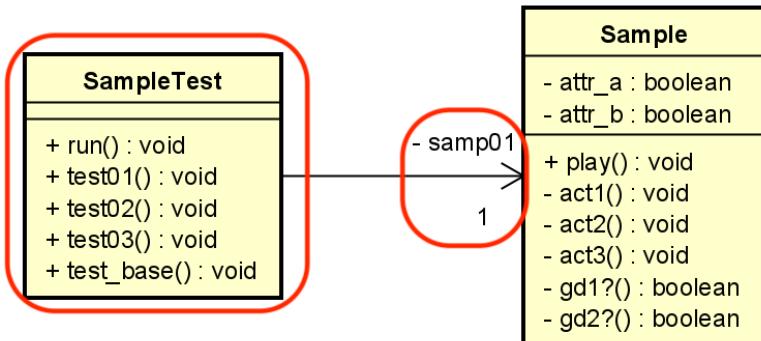


図 3.22 「SampleTest」クラスと関連を追加する

これで、クラス図とステートマシン図とRubyのコードの対応づけができました。実際の問題について、クラス図やステートマシン図を作成するときは、このようなルールを使う前提で作成します。そして、Rubyでコードを作成するときは、作成した図と対応づけのルールを使ってコードを作成します。

【参考】列挙型(enum)をクラス図に反映する方法について

Rubyにはenumのような列挙子を直接定義する方法がないので、このチュートリアルの簡単のために、状態を表す定数を定義するために「シンボル(:ST0など)」を使いました。

astah* は、Java、C#、C++については、enumを設定する方法を提供していますので、その方法で代用してもよいでしょう。

詳しい設定方法については、次の記事を参考にしてみてください。

Java、enumの設定

http://astah-users.change-vision.com/ja/modules/xhnewbb/viewtopic.php?topic_id=1249

3.2 変換ルールを使ったコードを確認する

検討用のクラス図とステートマシン図に対応づけたRubyのプログラムの動作を確認してみましょう。

3.2.1 検討用モデルに対応したRubyプログラムの全体

作成したプログラムの全体をリスト3.8に示します。このプログラムのコードには、「SampleTest」のインスタンスを作成して、テストを起動する処理や、テストメソッドの具体例が含まれています。テストクラスやテスト用メソッドの細かい説明は省きます。

リスト3.8 【Ruby】stm_sample.rb

```

1 # frozen_string_literal: true ①
2
3 # ステートマシン図とコードの対応づけルールの検討用クラス
4 class Sample
5   attr_accessor :attr_a, :attr_b ②
6
7   def initialize
8     @state = :ST0
9     @attr_a = true
10    @attr_b = true
11  end
12
13  def st0_proc(evt, param)
14    case evt
15    when :ev1
16      if gd1? # guard
17        puts "  gd1: #{gd1?}"
18        act1(evt, param)
19        @state = :ST1
20    else

```

```
21      puts "    <<< gd1: #{gd1?}, transition is ignored. >>>"  
22    end  
23  when :ev3  
24    act3(evt, param)  
25    @state = :ST2  
26  end  
27 end  
28  
29 def st1_proc(evt, param)  
30  case evt  
31  when :ev2  
32    act2(evt, param)  
33    puts "    gd2: #{gd2?}"  
34    if gd2? # guard on choice.  
35      @state = :ST2  
36    else  
37      act3(evt, param)  
38      @state = :ST0  
39    end  
40  when :ev3  
41    act3(evt, param)  
42    @state = :ST0  
43  end  
44 end  
45  
46 def play(evt, param)  
47  puts "#{@state} ->"  
48  puts "  event:#{evt}, param: #{param}"  
49  case @state  
50  when :ST0  
51    st0_proc(evt, param)  
52  when :ST1  
53    st1_proc(evt, param)  
54  when :ST2  
55    # none  
56  end  
57  puts "      -> #{@state}"  
58  puts 'finished.' if @state == :ST2  
59 end  
60  
61 private  
62  
63 def act1(evt, prm)  
64  puts "    act1: event:#{evt}, param: #{prm}"  
65 end  
66  
67 def act2(evt, prm)  
68  puts "    act2: event:#{evt}, param: #{prm}"  
69 end  
70  
71 def act3(evt, prm)  
72  puts "    act3: event:#{evt}, param: #{prm}"  
73 end  
74
```

```

75  def gd1?
76    @attr_a && @attr_b
77  end
78
79  def gd2?
80    !@attr_a || @attr_b
81  end
82 end
83
84 # 検討用クラスのテストケースを提供するクラス
85 class SampleTest ③
86   def test_base(events, data)
87     samp01 = Sample.new
88     samp01.attr_a = data[0]
89     samp01.attr_b = data[1]
90     events.each do |evt|
91       samp01.play(evt, Time.now.usec)
92     end
93     puts '====='
94   end
95
96  def test01
97    test_base(%i[ev1 ev2], [true, true])
98  end
99
100 def test02
101   test_base(%i[ev1 ev3], [false, false]) # :ev1 will be ignored.
102 end
103
104 def test03
105   test_base(%i[ev1 ev2 ev3], [true, false])
106 end
107
108 def run
109   test01
110   test02
111   test03
112 end
113 end
114
115 if $PROGRAM_NAME == __FILE__ ④
116   test = SampleTest.new
117   test.run
118 end

```

- ① 文字列の破壊的変更の検出を指示するマジックコメント。
- ② 簡便にテストするために、アクセサーを追加した(あくまでこのサンプルにおける便宜として)。
- ③ Sampleクラスのメソッドを呼び出すテストを定義したテスト用のクラスの定義。
- ④ テスト用クラスを実行するコード。このブロックはファイル名と実行プログラム名が一致するときに有効になる(そのときだけ実行される)。

3.2.2 検討用モデルに対応したRubyプログラムを動かす

コマンドプロンプトを起動して(MacやLinuxならターミナルを起動して)、プログラムを実行してみます(リスト3.9)。

動作させると、状態遷移やイベントが表示されます。作成したモデル図を比較して期待した動作をしているか確認してみましょう。

リスト3.9【端末】stm_sample.rbを実行する

```
C:\Users\kuboaki>cd Desktop\BowlingScore

C:\Users\kuboaki\Desktop\BowlingScore>ruby stm_sample.rb
ST0 ->
  event:ev1, param: 676481
    gd1: true
      act1: event:ev1, param: 676481
        -> ST1
ST1 ->
  event:ev2, param: 678623
    act2: event:ev2, param: 678623
    gd2: true
      -> ST2
finished.
=====
ST0 ->
  event:ev1, param: 680596
    <<< gd1: false, transition is ignored. >>>
      -> ST0
ST0 ->
  event:ev3, param: 681783
    act3: event:ev3, param: 681783
      -> ST2
finished.
=====
ST0 ->
  event:ev1, param: 683477
    <<< gd1: false, transition is ignored. >>>
      -> ST0
ST0 ->
  event:ev2, param: 685784
    -> ST0
ST0 ->
  event:ev3, param: 708214
    act3: event:ev3, param: 708214
      -> ST2
finished.
=====

C:\Users\kuboaki\Desktop\BowlingScore>
```

3.3 まとめ

クラス図とステートマシン図で動作を表現すれば、Rubyのコードに変換できることがわかりました。

3.3.1 構造のモデルを作成した手順

構造のモデルに登場する構成要素や要素間の関連を見つけ出し、スコアシートの構造を整理しました。

スコアシートの構造のモデルを作成した手順

1. スコアシートに関する構成要素の発見
 - 具体的なスコアシートを観察して、そこにある要素を洗い出してオブジェクト図で表した。
2. スコアシートに関するクラス図の作成
 - オブジェクト図に登場したオブジェクトからクラスを、オブジェクト同士のリンクから関連を見出し、クラス図に表した。

3.3.2 構造のモデルだけではプログラムは作れない

構造のモデルを作ったことで、プログラムを構成する要素と、要素同士の関係(関連)を表せるようになりました。また、クラスのインスタンスをオブジェクト図に見合うように作成すれば、実際に観察したスコアシートと同じようなデータ構造を作成できることもわかりました。

ところで、作りたかったのは、ボウリングのゲームスコアをつけるプログラムでした。つまり、プレーヤーがフレームごとに交代して投球し、そのたびにスコアを更新する処理です。ですが、これまでに作成した構造のモデルでは、ゲームの進行に応じてどんなことを処理するか表せていません。そのためには、処理の中身を表す「振る舞いのモデル」を作成する必要があります。次の章では、ステートマシン図を使って、フレームやスコアの振る舞いを表すモデルを作成します。

4 スコアやフレームの状態を調べる

設計で作成するモデルと実装に使うプログラムの対応づけができたました。この対応づけを前提に、ボウリングスコアのモデルの作成を進めましょう。

4.1 フレームの状態について検討する

図 2.1 を見ると、フレームの表示は、ゲームの進行状況によって変わっています。

ゲームの進行状況によってフレームの表示は異なる

- ・まだプレーしていないフレーム
- ・1投目の投球を待っているフレーム(現在のプレーヤーの現在のフレームの1投目の前)
- ・2投目の投球を待っているフレーム(現在のプレーヤーの現在のフレームの2投目の前)
- ・2投目が投球されて獲得ピン数が確定したフレーム(1,2投目のピン数とトータルスコアが表示されている)
- ・ストライクのボーナスが確定しないフレーム(ストライクの記録だけでトータルスコアは表示されていない)
- ・スペアのボーナスが確定しないフレーム(1投目とスペアの記録だけでトータルスコアは表示されていない)
- ・スペアまたはストライクのボーナスが確定したフレーム(1,2投目のピン数とトータルスコアが表示されている)

それについて、もう少し詳しく見てみましょう。

4.1.1 まだプレーしていないフレーム

まだプレーしていないフレームは、フレームの最初の状態です(図 4.1)。このフレームは、ピン数の入力を行っていません。

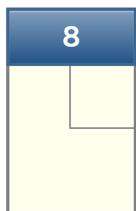


図 4.1 まだプレーしていないフレーム

4.1.2 1投目の投球を待っているフレーム

現在プレー中のフレームで、まだ1投目が投球されていない状態のフレームです(図 4.2)。次にピン数を受け取ると、このフレームの1投目に記録されます。

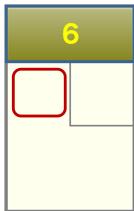


図 4.2 1投目の投球を待っているフレーム

4.1.3 2投目の投球を待っているフレーム

現在プレー中のフレームで、1投目がストライクでなかったときに2投目を待っている状態のフレームです(図 4.3)。次にピン数を受け取ると、このフレームの2投目に記録されます。



図 4.3 2投目の投球を待っているフレーム

4.1.4 スペアのボーナスの確定待ちのフレーム

現在プレー中のフレームの前のフレームがスペアで、現在のフレームが1投目の投球を待っているとき、前のフレームはスペアボーナスの確定待ちの状態です(図 4.4)。次にピン数を受け取ると、現在のフレームの1投目に記録されるとともに、前のフレームのスペアボーナスが確定します。

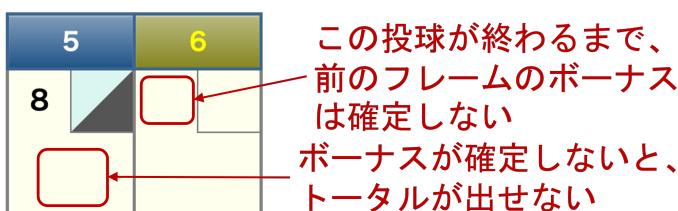


図 4.4 スペアのボーナスの確定待ちのフレーム

4.1.5 ストライクのボーナスの確定待ちのフレーム

ストライクボーナスの確定待ちには2通りの場合があります。

1つ目は、現在プレー中のフレームの前のフレームがストライクで、現在のフレームが2投目の投球を待っているときです。このとき、前のフレームはストライクボーナスの確定待ちの状態です(図 4.5)。次にピン数を受け取ると、現在のフレームの2投目に記録されるとともに、前のフレームのストライクボーナスが確定します。



図 4.5 ストライクのボーナスの確定待ちのフレーム(次がストライクでない)

2つ目は、現在プレー中のフレームが1投目の投球を待っていて、前のフレームと前の前のフレームがともにストライクであったとき(ダブルのとき)ときです。このとき、前の前のフレームはストライクボーナスの確定待ちの状態です(図 4.6)。次にピン数を受け取ると、現在のフレームの1投目に記録されるとともに、前の前のフレームのストライクボーナスが確定します。

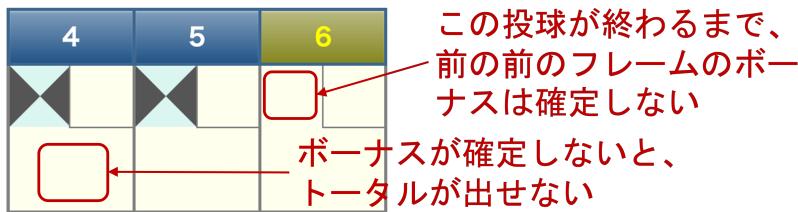


図 4.6 ストライクのボーナスの確定待ちのフレーム(次がストライク)

4.1.6 獲得ピン数とボーナスが確定したフレーム

現在のフレームがスペアやストライクにならなかったとき、そのフレームはボーナスの確定待ちにならず、この段階でそのフレームのトータル(ボーナスなしの獲得ピン数)が確定します。

スペアボーナスの確定待ち、またはストライクボーナスの確定待ちのフレームは、後のフレームの投球によってボーナスが確定すると、フレームのトータルが確定します。このとき、確定待ちのフレームでは、そのフレームの獲得ピン数と確定したボーナスの合計がそのフレームのトータルです。

フレームのトータルが求められると、それ以前のフレームまでの「のべのトータル」にそのフレームのトータルを加算して、のべのトータルを更新します。そして、更新したのべのトータルがフレームの下部に記録されます(図 4.7)。このとき、確定待ちになっていたフレームもののべのトータルが更新され、その段階のゲームのトータルも更新されます。

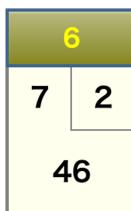


図 4.7 2投目を投球して、ピン数が確定したフレーム（のべのトータルが求められている）

4.2 フレームの状態をステートマシン図で表す

フレームの状態を検討した結果、フレームはゲームの状況によって変わる複数の状態を持つことがわかりました。また、フレームが表示できる情報も、状態によって異なることがわかりました。このことを、モデル図を使って表してみましょう。状態とその推移（状態遷移と呼びます）を表すには、ステートマシン図を使います。

4.2.1 「Frame」クラスにステートマシン図を追加する

「Frame」クラスの状態を表す図を描きたいので、「Frame」クラスに図を追加しましょう。

「Frame」クラスにステートマシン図を追加する

1. 「Frame」クラスにステートマシン図を追加する

- 構造ツリーから「Frame」クラスを選択し、右クリックしてポップアップメニューを表示する（図 4.8）。
- 「図の追加>ステートマシン図」でステートマシン図が追加される。

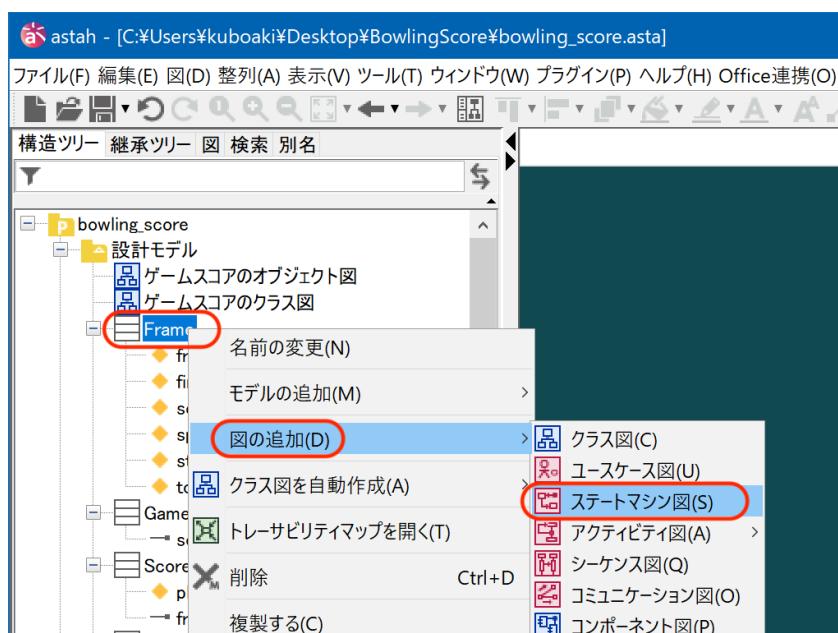


図 4.8 「Frame」クラスにステートマシン図を追加する

2. ステートマシン図に名前をつける

- 追加したステートマシン図のプロパティーの「ベース」を開く。
- 名前を編集して「Frameクラスのステートマシン図」とする。
- ダイアグラムエディタのタイトルやタブにも反映される。

4.2.2 フレームのステートマシン図を作成する

ステートマシン図に、フレームの状態と状態遷移を追加しましょう。

ステートマシン図に「Frame」クラスの状態遷移を作成する

1. 「Frame」クラスの最初の状態を作成する(図 4.9)。

- パレットから「開始疑似状態」を選択し、ステートマシン図に追加する。
- パレットから「状態」を選択し、ステートマシン図に追加する。
- 「まだプレーしていない」状態を「RESERVED」という名前にする。
- パレットから「遷移」を選択して、「開始疑似状態」から「RESERVED」へ遷移を引く。

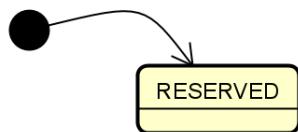


図 4.9 「Frame」クラスのステートマシン図に最初の状態を追加する

2. 最初の状態遷移とイベントを追加する(図 4.10)。

- 「1投目の投球を待っている」状態「BEFORE_1ST」を追加する。
- 「RESERVED」から「BEFORE_1ST」へ状態遷移を引き、イベント「SETUP」を割り当てる。

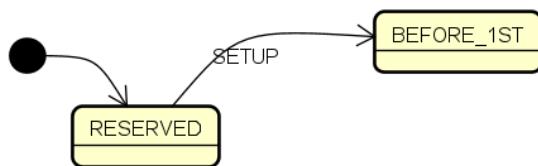


図 4.10 「Frame」クラスのステートマシン図に1投目の前の状態遷移とイベントを追加する

3. 1投目の後の状態遷移とイベントを追加する(図 4.11)。1投目の後は、受け取ったピン数によ

って遷移先は2通りある(ストライクか否か)ので「選択疑似状態」を使って遷移先を分ける。

- ・「2投目の投球を待っている」状態を「BEFORE_2ND」として追加する。
 - ・「スペアのボーナスの確定待ち」と「ストライクのボーナスの確定待ち」を「PENDING」として追加する。
 - ・「選択疑似状態」を追加する。
 - ・ピン数を受け取るイベント「PINS」(パラメーターとしてピン数 pins を持つ)を「BEFORE_1ST」から「選択疑似状態」への遷移に割り当てる。アクションとして、イベントで受け取ったピン数を1投目のピン数に保存する。
4. 選択疑似状態からの遷移を追加する。
- ・1投目のピン数がストライクであれば「PENDING」へ遷移する。このとき2投目のピン数は「0」にする。
 - ・1投目のピン数がストライクでなければ「BEFORE_2ND」へ遷移する。

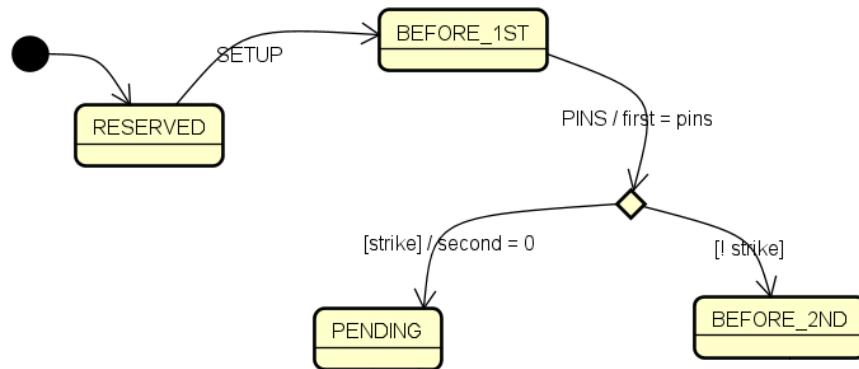


図 4.11 「Frame」クラスのステートマシン図に1投目の後の状態遷移とイベントを追加する

5. 2投目の後の状態遷移とイベントを追加する(図 4.12)。2投目の後は、受け取ったピン数によって遷移先は2通りある(スペアか否か)ので「選択疑似状態」を使って遷移先を分ける。
- ・「獲得ピン数とボーナスが確定した」状態を表す「FIXED」として追加する。
 - ・2投目の後の処理の選択のために「選択疑似状態」を追加する。
 - ・「BEFORE_2ND」でイベント「PINS」を受け取ると、選択疑似状態へ遷移する。アクションとして、受け取ったピン数を2投目のピン数に保存する。
6. 選択疑似状態からの遷移を追加する。
- ・2投目のピン数がスペアであれば「PENDING」へ遷移する。
 - ・2投目のピン数がスペアでなければ「FIXED」へ遷移する。
7. 「PENDING」からの遷移を追加する。
- ・「PENDING」中のフレームで、イベント「DETERMINE」を受け取ったらトータルが確定したとし、「FIXED」へ遷移する。

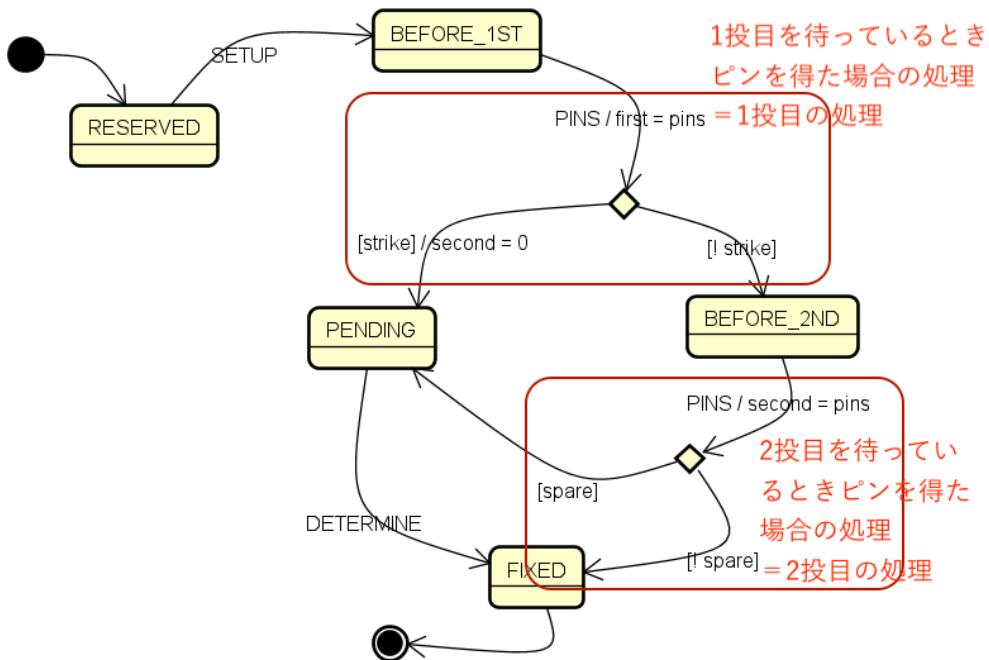


図 4.12 「Frame」クラスのステートマシン図に2投目の後の状態遷移とイベントを追加する

これで、フレームのステートマシン図が作成できました。

4.2.3 フレームのクラス図を更新する

ステートマシン図で状態遷移のために追加したアクションやガード条件用の処理を「Frame」クラスのメソッドに追加しておきましょう(図 4.13)。

Frame	
- frame_no : int	
- first : int	
- second : int	
- spare_bonus : int	
- strike_bonus : int	
- total : int	
+ action() : void	
+ before_1st_proc() : void	
+ before_2nd_proc() : void	
+ strike?() : boolean	
+ spare?() : boolean	
+ miss?() : boolean	
+ gutter?() : boolean	
+ fixed?() : boolean	
+ to_s() : String	

図 4.13 ステートマシン図に合わせて「Frame」クラスを更新する

4.2.4 フレームの処理をプログラムに変換する

フレームのクラスとステートマシン図が作成できたので、Rubyのプログラムに変換してみましょう。「3」で決めたルールにしたがって、モデルからコードへ変換します。

まず、プログラムの初期化とアクションの部分は、リスト 4.1 のようになるでしょう。

リスト 4.1 【Ruby】score.rb(1)

```
# frozen_string_literal: true

require 'securerandom'

# Frameは1フレーム分のピン数やボーナスを記録する
class Frame
  attr_reader :frame_no
  attr_accessor :first, :second, :spare_bonus, :strike_bonus, :total, :state

  def initialize(frame_no)
    @frame_no = frame_no
    @first = 0
    @second = 0
    @spare_bonus = 0
    @strike_bonus = 0
    @total = 0
    @state = :RESERVED ①
  end

  def action(event, pins=0)
    case @state ②
    when :RESERVED
      case event
      when :SETUP ③
        @state = :BEFORE_1ST
      else
        puts "invalid event: #{event} is ignored."
      end
    when :BEFORE_1ST
      before_1st_proc(event, pins) ④
    when :BEFORE_2ND
      before_2nd_proc(event, pins) ⑤
    when :PENDING
      case event
      when :DETERMINE ⑥
        @state = :FIXED
      end
    when :FIXED
      puts 'fixed.'
    end
  end
end
```

- ① フレームの初期状態は「RESERVED」とする。状態はRubyのシンボルを使って表現する。
- ② 状態に応じて処理を分ける。
- ③ 「RESERVED」状態では「SETUP」イベントを受け取り、「BEFORE_1ST」状態へ遷移する。他のイベントが来たら無視する。
- ④ 「BEFORE_1ST」状態では「PINS」イベントを待つ。詳細な処理は「before_1st_proc」メソッドに記載する。

- ⑤ 「BEFORE_2ND」状態では「PINS」イベントを待つ。詳細な処理は「before_2nd_por」メソッドに記載する。
- ⑥ 「PENDING」状態では「DETERMINE」イベントを受け取り、「FIXED」状態へ遷移する。

ガード条件やアクションのメソッドは、リスト 4.2 のようになるでしょう。

リスト 4.2 【Ruby】score.rb(2)

```
# frozen_string_literal: true

require 'securerandom'

# Frameは1フレーム分のピン数やボーナスを記録する
class Frame
    # initialize、actionの定義がここにある

    def frame_score ❶
        @first + @second + @spare_bonus + @strike_bonus
    end

    def strike? ❷
        @first == 10
    end

    def spare?
        @first < 10 && (@first + @second) == 10
    end

    def miss?
        @first < 10 && @second.zero? && @state == :FIXED
    end

    def gutter?
        @first.zero?
    end

    def fixed? ❸
        @state == :FIXED
    end

    def to_s ❹
        total = if @state == :FIXED
                  @total
                else
                  ''
                end
        format '|%2d|%3s|%3s|%5s|%11d|%11d|%12d|%11s|',
               @frame_no, @first, @second, total, frame_score,
               @spare_bonus, @strike_bonus, @state
    end

    private ❺
```

```

def before_1st_proc(evt, pins) ⑥
  case evt
  when :PINS
    puts "invalid pins: #{pins}" if pins.negative? || pins > 10
    @first = pins
    @state = if strike?
      @second = 0
      :PENDING
    else
      :BEFORE_2ND
    end
  else
    puts "invalid event: #{evt} on #{@state}."
  end
end

def before_2nd_proc(evt, pins) ⑦
  case evt
  when :PINS
    puts "invalid pins: #{pins}" if pins.negative? || pins > (10 - @first)
    @second = pins
    @state = if spare?
      :PENDING
    else
      :FIXED
    end
  else
    puts "invalid event: #{evt} on #{@state}."
  end
end

```

- ① フレームのスコアを計算するメソッド
- ② ストライクかどうか判定するガード条件用のメソッド。
- ③ フレームのスコアが確定したか調べるメソッド。
- ④ 「Frame」クラスのインスタンスを文字列化するメソッド。呼び出し時点の「Frame」クラスが保持するインスタン変数の値を出力するのに使う。
- ⑤ これ以降のメソッドはプライベート。
- ⑥ 「1投目の投球を待っている」状態を担当するメソッド。イベント「PINS」を受け取り、1投目のピン数に保存する。その後ストライクかどうか調べ、次の状態へ遷移する。
- ⑦ 「2投目の投球を待っている」状態を担当するメソッド。イベント「PINS」を受け取り、2投目のピン数に保存する。その後スペアかどうか調べ、次の状態へ遷移する。

4.3 サービスフレームの扱いについて検討する

「クラシックスコアリング」の場合、第10フレームが他のフレームとは異なっています。サービスフレームという考え方があり、ストライクやスペアの場合に追加で投球できます。

このサービスフレームの扱い方にについて整理しておきましょう。

4.3.1 第10フレームがストライクもスペアもない場合

第10フレームがストライクでもスペアもない場合、サービスフレームは提供されません。第9フレームまでの通常のフレームと同様、1投目と2投目を記録するだけです(図4.14)。

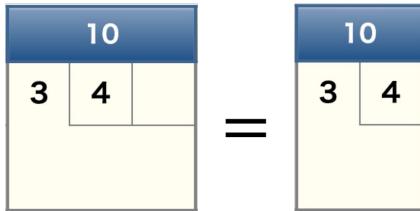


図4.14 第10フレームがストライクもスペアもない場合

4.3.2 第10フレームがスペアの場合

第10フレームの2投目でスペアになった場合、1投目と2投目を通常のフレームと同様に記録したのち、もう1投追加されます。これを、第10フレームに加えて、第11フレームの1投目が追加されたとみなします。つまり、ゲームスコアのデータを「図4.15」のように構成します。

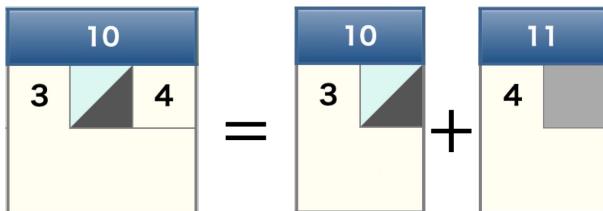


図4.15 第10フレームがスペアの場合(第11フレームの2投目はない)

スコアをこのように構成しておくと、第10フレームの場合も、通常フレームの組み合わせによってボーナスが計算できます。ゲーム終了時の第10フレームのスコア(ピン数に以後の投球によるボーナスを加えたスコア)を取得すると、それがゲームのスコアです。

4.3.3 第10フレームの1投目がストライクの場合

第10フレームの1投目がストライクの場合、第10フレームをストライクとした上で、もう2投追加されます。これを、第10フレームに加えて、第11フレームの1投目と2投目が追加されたとみなします。つまり、ゲームスコアのデータを「図4.16」のように構成します。ただし、第11フレームの1投目がストライクの場合は、同じ2投追加でも次の「2投目もストライクの場合」の考え方を使います。

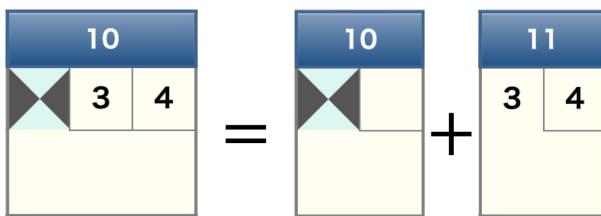


図 4.16 第10フレームの1投目がストライクの場合

スコアをこのように構成しておくと、第10フレームの場合も、通常フレームの組み合わせによってボーナスが計算できます。ゲーム終了時の10フレームのスコア(ピン数に以後の投球によるボーナスを加えたスコア)を取得すると、それがゲームのスコアです。

4.3.4 第10フレームの2投目もストライクの場合

第10フレームの2投目もストライクの場合、第10フレーム、第11フレームをストライクとした上で、もう1投追加されます。これを、第10フレーム、第11フレームに加えて、第12フレーム目の1投目が追加されたとみなします。つまり、ゲームスコアのデータを「図 4.17」のように構成します。このように構成すれば、通常のフレームでダブルをとったときの計算方法(ストライクが続いたときはさらに次のフレームの1投目が2投目として加算される)のままで第10フレームのボーナスが計算できます。

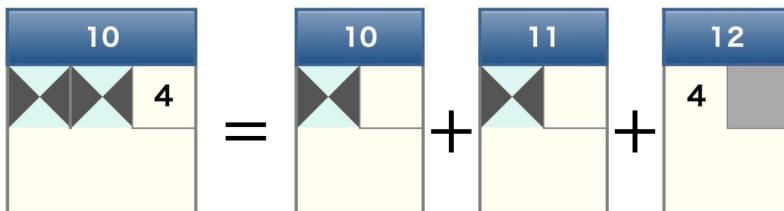


図 4.17 第10フレームの2投目もストライクの場合

スコアをこのように構成しておくと、第10フレームの場合も、通常フレームの組み合わせによってボーナスが計算できます。ゲーム終了時の10フレームのスコア(ピン数に以後の投球によるボーナスを加えたスコア)を取得すると、それがゲームのスコアです。

4.4 スコアの状態をステートマシン図で表す

サービスフレームの扱い方を検討した結果、工夫すれば通常のフレームと同じように扱えることがわかりました。それでは、フレームの集まりであるスコアについて、どのような処理をすればよいのか検討しましょう。

4.4.1 「Score」クラスにステートマシン図を追加する

「Frame」クラスにステートマシン図を追加したのと同じ手順で、「Score」クラスにステートマシン図を追加します(図 4.18)。

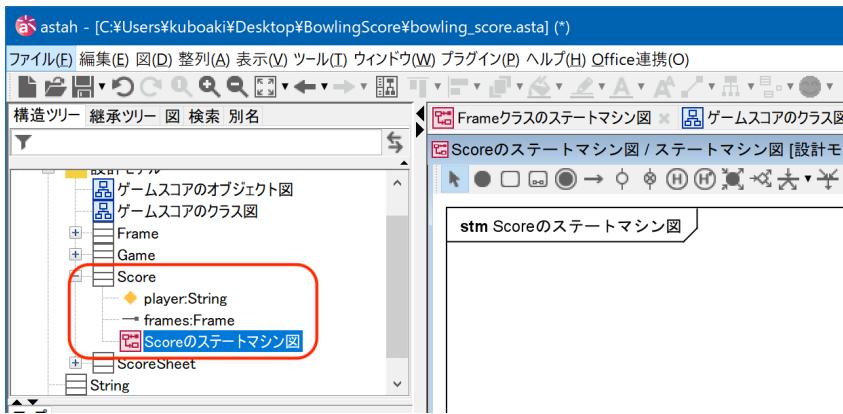


図 4.18 「Score」クラスにステートマシン図を追加する

4.4.2 スコアのステートマシン図を作成する

ステートマシン図に、検討した結果を使って、スコアの状態と状態遷移を追加しましょう。

ステートマシン図に「Score」クラスの状態遷移を作成する

1. 「Score」クラスの状態を追加する(図 4.19)。

- パレットから「開始疑似状態」をステートマシン図に追加する。
- パレットから「状態」を選択し、ステートマシン図に追加する。
- 「1投目待ち」の状態として状態名を「WAIT_FOR_1ST」に設定する。
- 「2投目待ち」の状態として状態名を「WAIT_FOR_2ND」に設定する。
- 「ゲームの終了」の状態として状態名を「FINISHED」に設定する。
- パレットから「終了疑似状態」をステートマシン図に追加する。

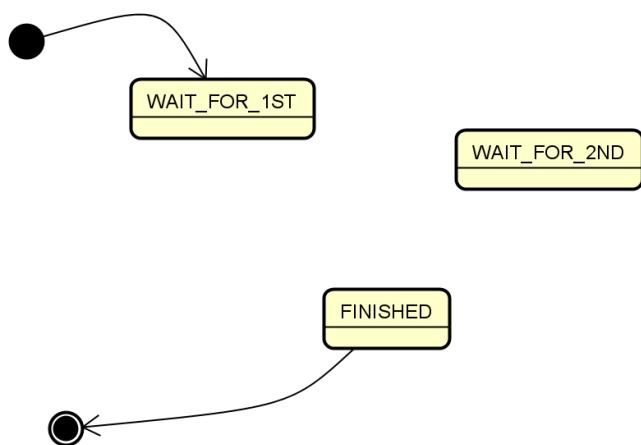


図 4.19 ステートマシン図に「Score」クラスの状態を追加する

2. 「Score」クラスのイベントとアクションを作成する(図 4.20)。

3. 「WAIT_FOR_1ST」からの遷移には、ストライクの場合、ストライクでない場合、ゲームが終了の場合があるので、「選択疑似状態」を追加する。

- ・「WAIT_FOR_1ST」から「選択疑似状態」への遷移では、ピン数を受け取るのを待っている。受け取ったときには、現在にフレームヘピン数のイベントを送る。その後、1投目の後のスペアとストライクのボーナスを計算し、のべのトータルを更新する。
 - ・「選択疑似状態」から「WAIT_FOR_1ST」への遷移のガード条件は「ストライクである」こと。このときはアクションとして「次のフレームへ進む」。
 - ・「選択疑似状態」から「WAIT_FOR_2ND」への遷移のガード条件は「ストライクでない（かつゲーム終了ではない）」こと。
 - ・「選択疑似状態」から「FINISHED」への遷移のガード条件は「ゲーム終了である」であること。
4. 「WAIT_FOR_2ND」からの遷移には、ゲーム終了の場合とそうでない場合があるので、「選択疑似状態」を追加する。
- ・「WAIT_FOR_1ST」から「選択疑似状態」への遷移では、ピン数を受け取るのを待っている。受け取ったときには、現在にフレームヘピン数のイベントを送る。そして、2投目の後のスペアとストライクのボーナスを計算し、のべのトータルを更新する。
 - ・「選択疑似状態」から「WAIT_FOR_1ST」への遷移のガード条件は「ゲーム終了ではない」こと。このときはアクションとして「次のフレームへ進む」。
 - ・「選択疑似状態」から「FINISHED」への遷移のガード条件は「ゲーム終了である」であること。ゲーム終了は、10フレーム目の状態が「FIXED」になったことで判定できる(そのようなメソッドを用意する)。

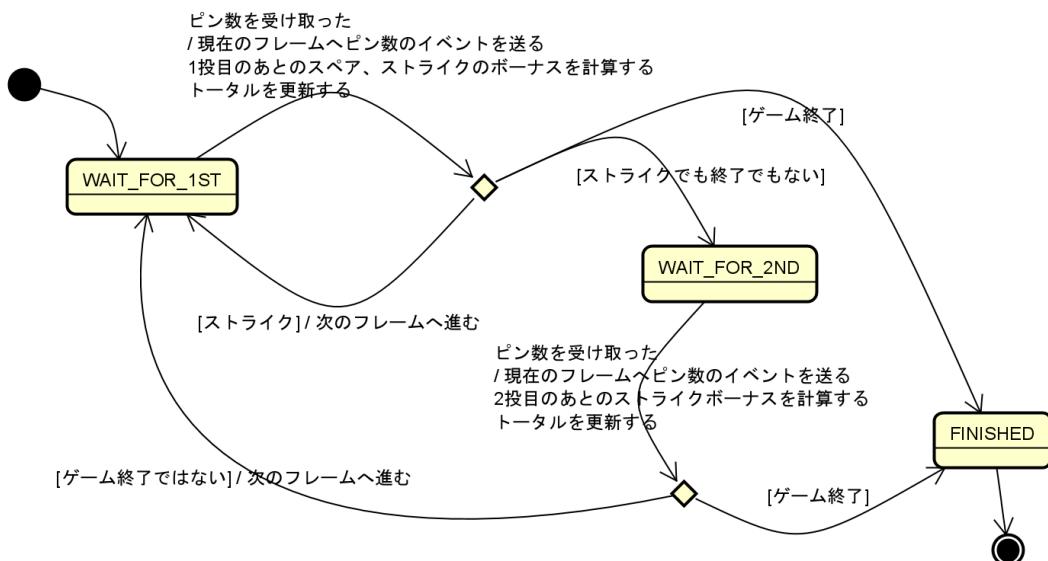


図 4.20 ステートマシン図に「Score」クラスのイベントとアクションを追加する

4.4.3 クラス図に検討結果を反映する

第10フレームの場合にもサービスフレームのために追加のフレームを用意することで、通常フレームと同じようにピン数やボーナスを扱えるようになりました。

クラス図にこの結果を反映しましょう。

ステートマシン図に合わせてクラス図を更新する(図 4.21)

1. 「Frame」クラス側の関連端の多重度を「10」から「12」に変更する。
 - 。これは、通常のフレームを追加する方法でサービスフレームを処理するための追加。
2. 関連にノートをつけて説明をつけておく。
 - 。パレットからノートを選択し、クラス図に追加する。
 - 。ノートに「Frame側の多重度は、サービスフレーム用に必要なフレーム分だけ追加してある」という説明を追加する。
 - 。ノートから関連の線に向かってアンカーを引く。
 - 。パレットからノートのアンカーを選択し、ノートにマウスカーソルを移動して青枠が表示されるのを待つ。
 - 。マウスのボタンを押したまま、マウスカーソルをドラッグし、関連の線に近づけ青枠が表示されるのを待つ。
3. 「Score」クラスにステートマシン図で作成したメソッドを追加しておく。
 - 。「次のフレームへ進む」メソッド「go_next_frame」を追加する。
 - 。「ゲーム終了か判定する」メソッド「finished?」を追加する。
 - 。「1投目の後のスペアのボーナスを計算する」メソッド「calc_spare_bonus_after_1st」を追加する。
 - 。「1投目の後のストライクのボーナスを計算する」メソッド「calc_strike_bonus_after_1st」を追加する。
 - 。「2投目の後のストライクのボーナスを計算する」メソッド「calc_strike_bonus_after_2st」を追加する。
 - 。「のべのトータルを更新する」メソッド「update_total」を追加する。
 - 。スコアを記録するメソッド(ステートマシン図の処理を担当する)メソッド「scoring」を追加する。
 - 。ステートマシン図の状態ごとの処理を担当するメソッド「wait_for_1st_proc」と「wait_for_2nd_proc」を追加する。
 - 。スコアの記録を文字列化するメソッド「to_s」を追加する。
4. 関連を追加する
 - 。「Score」クラスから「Frame」クラスへ、「現在のフレーム」を指す関連「current」を追加する。

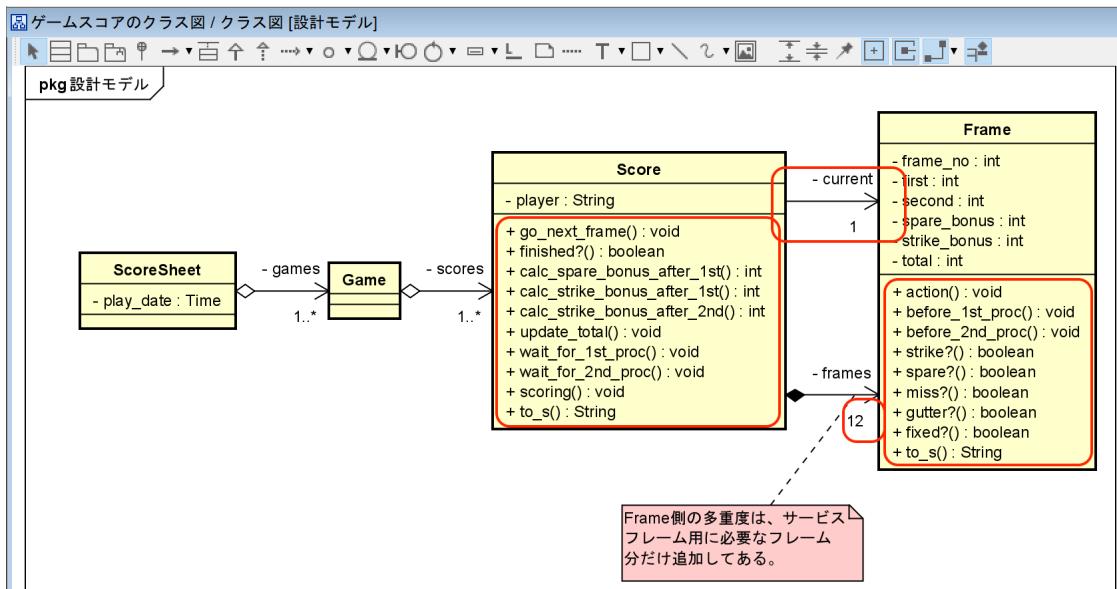


図 4.21 ステートマシン図に合わせてクラス図を更新する



ノート内の文章を編集するときは、ノートを選択した状態でプロパティを使って編集すると、改行が入力しやすくなります。

4.4.4 スコアの処理をプログラムに変換する

スコアのクラスとステートマシン図が作成できたので、Rubyのプログラムに変換してみましょう。変換したプログラムの初期化やユーティリティメソッドの部分は、リスト 4.3 のようになるでしょう。

リスト 4.3 【Ruby】score.rb(3)

```
# frozen_string_literal: true

require 'securerandom' ①

# Frameクラスの定義がここにある

# スコアは各人の10フレーム分のスコアを記録する
class Score ②
  attr_accessor :id, :player, :fno, :frames, :state

  def initialize(name)
    @id = SecureRandom.urlsafe_base64(8) ③
    @player = name
    @fno = 1
    @frames = []
    (-1..13).each do |fno| # (-1, 0) are dummy frame ④
      @frames.append Frame.new(fno)
    end
    @state = :WAIT_FOR_1ST ⑤
    @frames[fno2idx(@fno)].action(:SETUP) ⑥
  end

  def fno2idx(fno) ⑦
    fno + 1 # frame number 1 => array index 3 (0 origin).
  end

  def frame(fno)
    @frames[fno2idx(fno)] # return index on @frames at frame number.
  end

  def go_next_frame ⑧
    @fno += 1
    @frames[fno2idx(fno)].action(:SETUP)
  end

  def current ⑨
    frame(@fno)
  end
end
```

- ① 安全なIDを生成するためのライブラリをインポートした。
- ② 「Score」クラスの定義のはじまり。
- ③ スコアのインスタンスにIDをつけておく。
- ④ 「Frame」のインスタンスの作成。クラス図では、「Score」から「Frame」へのコンポジションとして表されている部分。設計上は12個だが、サービスフレームで「次」を参照する場面、第1フレームで「前のフレーム」「前の前のフレーム」を参照する場面で参照するダミーのフレームを追加している。
- ⑤ スコアの最初の状態を「WAIT_FOR_1ST」にする。

- ⑥ 第1フレームへ「SETUP」イベントを送る。
- ⑦ フレーム番号とフレームの配列のインデックスを対応づけるメソッド(ダミーのフレームを第1フレームの前に追加したためのオフセット)。
- ⑧ 現在のフレームを「次のフレームへ進める」メソッド。このメソッドでは、次のフレームに「SETUP」イベントを送る。
- ⑨ 「現在のフレーム」を参照するためのメソッド。クラス図では関連端名が「current」の「Frame」クラスへの関連で表されている。

ボーナス計算やトータル計算のメソッドは [リスト 4.4](#) のようになるでしょう。

リスト 4.4 【Ruby】score.rb(4)

```
# Frameクラスの定義やそれ以前のコードがここにある

# スコアは各人の10フレーム分のスコアを記録する
class Score
    # アクセサーの定義がここにある
    # initialize、fno2idx、frame、go_next_frame、currentメソッドの定義がここにある

    def prev ①
        frame(@fno - 1)
    end

    def pprev ②
        frame(@fno - 2)
    end

    def calc_spare_bonus_after_1st ③
        return unless prev.spare?

        prev.spare_bonus = current.first
        prev.action(:DETERMINE)
    end

    def calc_strike_bonus_after_1st ④
        return unless prev.strike? && pprev.strike?

        pprev.strike_bonus = prev.first + current.first
        pprev.action(:DETERMINE)
    end

    def calc_strike_bonus_after_2nd ⑤
        return unless prev.strike?

        prev.strike_bonus = current.first + current.second
        prev.action(:DETERMINE)
    end

    def update_total ⑥
        @frames.each_cons(2) do |prev, cur|
            cur.total = prev.total + cur.frame_score
        end
    end

    def finished? ⑦
        frame(10).fixed?
    end
end
```

① ボーナス計算で使う、「前のフレーム」を得るメソッド。

② ボーナス計算で使う、「前の前のフレーム」を得るメソッド。

- ③ 1投目の後のスペアボーナスを計算するメソッド。前のフレームのスコアが確定するので、前のフレームへ「DETERMINE」イベントを送る。
- ④ 1投目の後のストライクボーナスを計算するメソッド。前の前のフレームからストライクが続いていた場合、ここでスコアが確定するので、前の前のフレームへ「DETERMINE」イベントを送る。
- ⑤ 2投目の後のストライクボーナスを計算するメソッド。前のフレームのスコアが確定するので、前のフレームへ「DETERMINE」イベントを送る。
- ⑥ 「のべのトータル」を更新するメソッド。each_consは、配列から指定した数ずつ要素を取り出すメソッド。
- ⑦ 「ゲーム終了」の判定用のメソッド。サービスフレームの検討結果から、第10フレームが「FIXED」になれば、そのゲームは終了とみなせる。

そして、ステートマシン図で表した、実際にゲームの進行に合わせてスコアを記録する処理をするメソッドは [リスト 4.5](#) のようになるでしょう。

リスト 4.5 【Ruby】score.rb(5)

```
# Frameクラスの定義やそれ以前のコードがここにある

# スコアは各人の10フレーム分のスコアを記録する
class Score
    # アクセサーの定義がここにある
    # initializeからfinished?までのメソッドの定義がここにある

    def wait_for_1st_proc(pins) ①
        current.action(:PINS, pins) ②
        calc_spare_bonus_after_1st
        calc_strike_bonus_after_1st
        update_total
        if finished? ③
            @state = :FINISHED
        elsif current.strike? ④
            @state = :WAIT_FOR_1ST
            go_next_frame
        else ⑤
            @state = :WAIT_FOR_2ND
        end
    end

    def wait_for_2nd_proc(pins) ⑥
        current.action(:PINS, pins) ⑦
        calc_strike_bonus_after_2nd
        update_total
        if finished? ⑧
            @state = :FINISHED
        else ⑨
            @state = :WAIT_FOR_1ST
            go_next_frame
        end
    end

    def scoring(pins) ⑩
        case @state
        when :WAIT_FOR_1ST
            wait_for_1st_proc(pins)
        when :WAIT_FOR_2ND
            wait_for_2nd_proc(pins)
        when :FINISHED
            puts 'finished'
        end
    end

    def to_s ⑪
        "Player:#{@player}, Score(id: #{@id}), Frame:#{@fno},
|No|1st|2nd|Total|Frame Score|Spare Bonus|Strike Bonus|Frame State|
#{@frames.join("\n")}"
    end
end
```

- ① 「WAIT_FOR_1ST」状態のときの状態遷移のメソッド。
- ② 現在のフレームについて、ピン数を更新し、ボーナスを計算し、のべのトータルを更新する。
- ③ 「選択疑似状態」での遷移先の分岐処理。「ゲーム終了」の場合は「FINISHED」へ遷移する。
- ④ 「ストライクだった」の場合は、次のフレームへ進んで「WAIT_FOR_1ST」へ遷移する。
- ⑤ それ以外のときは2投目を待つので、「WAIT_FOR_2ND」へ遷移する。
- ⑥ 「WAIT_FOR_2ND」状態のときの状態遷移のメソッド。
- ⑦ 現在のフレームについて、ピン数を更新し、ボーナスを計算し、のべのトータルを更新する。
- ⑧ 「選択疑似状態」での遷移先の分岐処理。「ゲーム終了」の場合は「FINISHED」へ遷移する。
- ⑨ それ以外のときは、次のフレームへ進んで「WAIT_FOR_1ST」へ遷移する。
- ⑩ スコアを記録するステートマシン図の振る舞いを担当するメソッド。状態に応じてそれぞれの状態用のメソッドを呼び出す。
- ⑪ 「スコア」クラスのインスタンスの内容(内包するフレームも含む)を文字列化するメソッド。

これで、フレームとスコアを、それぞれのステートマシン図で表した振る舞いに合わせて動作するプログラムに変換できました。

4.5 ゲームの進行について検討する

残るは、複数名のスコアのセットで構成される「Game」と、複数の「Game」を記録する「ScoreSheet」クラスです。

4.5.1 ヨーロピアン方式とアメリカン方式

ボウリングを複数名で楽しむとき、みなさんはたいてい、1組のチームで1つのレーンを使って、1フレームごとにプレーヤーが交代しながらプレイします。このようなゲームの方式は、「ヨーロピアン方式」と呼ばれています。

別的方式として、ボールラック(ボールが返ってくるラック)を挟んだ2つのレーンを、1フレーム交代で使ってプレーする方式があります。このようなゲームの方式は「アメリカン方式」と呼ばれています。

このチュートリアルでは、ヨーロピアン方式を使うことにします。

4.5.2 Gameクラスの処理

ほとんどのみなさんがボウリングをやるときにゲームを進行する手順は、「ヨーロピアン方式」と呼ばれています。「ヨーロピアン方式」による進行を想定して、「Game」クラスはどのような手順で動作させるべきか整理しましょう。

ボウリングのゲームを進める手順(ヨーロピアン方式)

1. スコアシートに参加するプレーヤー名を書く(エントリーする)。
2. 書いたプレーヤーの順に1フレーム分プレーする(各自のターン)。
3. 次のプレーヤと交代する(次のターンへ進む)。
4. 全員がゲーム終了するまで手順を繰り返す。

この方式に合わせてスコアを記録する処理をする「Game」クラスは、リスト 4.6 のようになるでしょう。

リスト 4.6 【Ruby】score.rb(6)

```
# FrameクラスとScoreクラスの定義やそれ以前のコードがここにある

# Gameクラスは複数名の1ゲーム分のスコアのセットを構成する
class Game
  attr_reader :id, :turn, :scores

  def initialize
    @id = SecureRandom.urlsafe_base64(8) ①
    @turn = 0
    @scores = []
  end

  def entry(name = 'unknown') ②
    @scores.append(Score.new(name))
  end

  def turn_player_name ③
    @scores[@turn].player
  end

  def go_next_turn ④
    @turn = (@turn + 1) % @scores.size
  end

  def playing(score_index, pins) ⑤
    @scores[score_index].scoring(pins)
    if @scores[score_index].fno > 10 ⑥
      go_next_turn if @scores[score_index].finished?
    elsif @scores[score_index].current.state == :BEFORE_1ST ⑦
      go_next_turn
    end
  end

  def finished? ⑧
    @scores.reject(&:finished?) == []
  end

  def to_s ⑨
    "Game(id:#{@id}),\n#{@scores.join("\n")}"
  end
end
```

- ① ゲームごとにユニークなIDをつけておく。
- ② ゲームに参加するプレーヤーを登録するメソッド。そのプレーヤーの今回のゲーム分のスコアも用意する。
- ③ 現在プレー中のプレーヤー名を取得するメソッド。
- ④ 次のプレーヤーに交代するメソッド。登録したプレーヤーの順に進み、最後まで来たら最初のプレーヤーへ戻る。

- ⑤ ゲームを実行するメソッド。現在のプレーヤーのスコアクラスのscoring メソッドを呼び出して1フレーム分のプレーを記録する。
- ⑥ サービスマネージャーでは、2フレーム以上プレーする場合があるので、そのときは交代しない。
- ⑦ 現在のプレーヤーの現在の状態が「BEFORE_1ST」なら、次のフレームの投球待ちになっているので、プレーヤーを交代する。
- ⑧ すべてのプレーヤーがゲーム終了かどうか調べるメソッド。
- ⑨ ゲームの状況を文字列化するメソッド。

クラス図にもこの結果を反映しておきましょう(図 4.22)。現在のプレーヤーは、関連端名が「turn」の関連によって参照しているScoreを使っています。

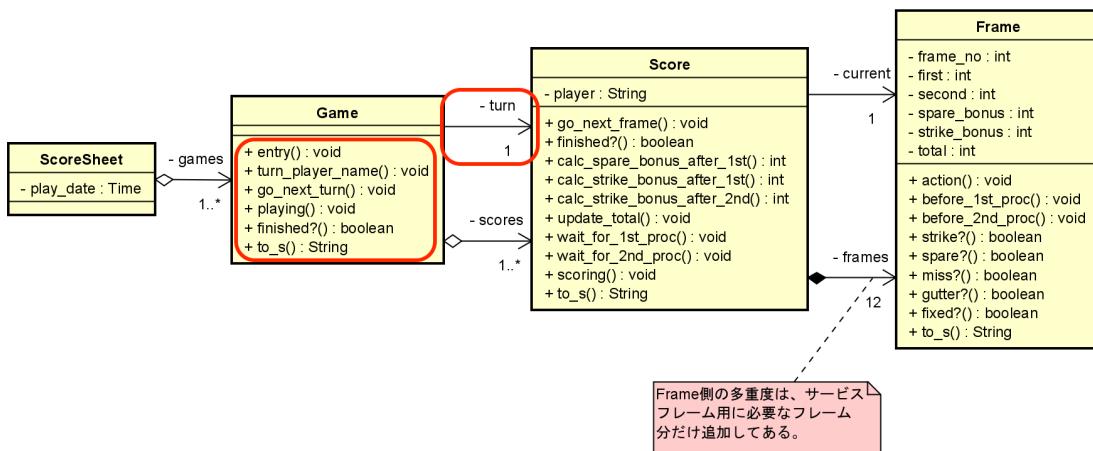


図 4.22 「Game」クラスを更新したクラス図

4.5.3 ScoreSheetクラスの処理

「Game」クラスが作成できたので、スコアシートを扱う「ScoreSheet」クラスも作成できそうですね。新しいスコアシートを作成して、そこに必要な数のゲームを追加すれば済みそうです。

この方式に合わせてスコアを記録する処理をする「Game」クラスは、リスト 4.7 のようになるでしょう。

リスト 4.7 【Ruby】score.rb(7)

```
# FrameクラスとScoreクラスとGameクラスの定義やそれ以前のコードがここにある

# ScoreSheetは、複数名の複数回のGameを記録する
class ScoreSheet
  attr_accessor :id, :time, :games

  def initialize(date) ①
    @id = SecureRandom.urlsafe_base64(8) ②
    @play_date = date
    @games = []
  end

  def add_game(games = 1) ③
    games.times do
      @games.append(Game.new)
    end
  end

  def to_s ④
    "Score Sheet Date: #{@time}(id:#{@id})"
  end
end
```

- ① スコアシートを作成するときは、作成時の日時を控えておく。
- ② スコアシートごとにユニークなIDをつけておく。
- ③ 希望する数のゲームをシートに追加するメソッド。引数がないときは1組だけ追加する。
- ④ スコアシートの状況を文字列化するメソッド。

クラス図にもこの結果を反映しておきましょう(図 4.23)。現在のプレーヤーは、関連端名が「turn」の関連によって参照しているScoreを使っています。

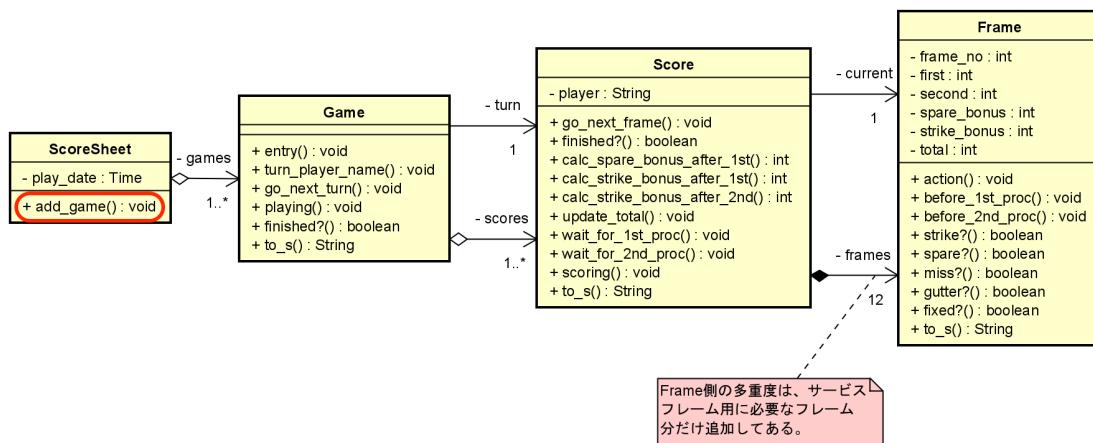


図 4.23 「ScoreSheet」クラスを更新したクラス図

4.6 まとめ

ボウリングのゲームスコアを記録するスコアシートの振る舞いを検討しました。

4.6.1 振る舞いのモデルを作成した手順

振る舞いのモデルに登場する構成要素(状態、イベント、アクション)を洗い出し、ゲームの手順を整理しました。

スコアシートの振る舞いのモデルを作成した手順

1. スコアに関する状態の発見
 - スコアが記録されるときのフレームの変化を観察して、フレームにどのような状態があるか洗い出した。
2. スコアに関するステートマシン図の作成
 - 洗い出したフレームの状態に、関連するイベントやアクションを考えてステートマシン図に表した。
3. フレームスコアに関する状態の発見
 - ゲームを進めるときのスコアの変化を観察して、スコアにどのような状態があるか洗い出した。
4. フレームに関するステートマシン図の作成
 - サービスフレームの動作を観察して、通常のフレームを追加してスコアを記録する方法を発見した。
 - 洗い出したスコアの状態に関連するイベントやアクションを考えてステートマシン図に表した。

4.6.2 モデルとコードの対応づけは振る舞い設計の前に

ステートマシン図を使うことで、フレームやスコアの動作を表せました。そして、あらかじめモデルとコードを対応づけておけば、それを前提として(モデルやコードの構成方式として)振る舞いの設計に活かせます。

モデルとコードが対応づけられていれば、Rubyのプログラムはステートマシン図から変換するように作成できます。つまり、解決すべき課題があったとき、その課題を構造のモデルと振る舞いのモデルで表すことができれば、そのモデルなりのプログラムが作成できるわけです。一方、このような方法で作成したプログラムが期待した動作をしない場合には、モデルが誤っているか、対応づけのルールに不備があるということです。

【参考】モデル変換とモデル駆動開発(MDD)

ここで示したようなモデルとコードの対応づけのほかに、モデルから別のモデルへ、あるいはコードから別のコードへといった対応づけも考えられます。モデルからコードを生成する方法には、このチュートリアルのように図の要素とコードの要素を対応させるルールを決めて手で変換する方法以外にもあります。例えば、コード片のテンプレート(スニペットなどと呼ばれます)にモデルのデータベースへの問い合わせを埋め込んだスクリプトを使って生成する方法はよく使われています

これらは、ソフトウェア設計における「モデル変換」と呼ばれています(コードも一種のモデル表現とみなせます)。そして、モデル変換を用いて開発プロセスにおける各工程間をモデルで接続して開発することを「モデル駆動開発(MDD: Model Driven Development)」と呼びます。

モデル駆動開発を紹介している記事として次の記事があります。

モデル駆動開発におけるモデル変換の役割

<https://codezine.jp/article/detail/10597>

この記事では、モデル変換の繰り返しによる開発方法であることや、その実施例を紹介しています。

5 できあがったプログラムを試す

対応づけルールを使って、モデルから変換して作成したプログラムができあがりました。できあがったプログラムを確認し、実際に動かしてみましょう。

5.1 プログラムを完成させる

まず、プログラムをテストできるよう、完成させましょう。

5.1.1 作成したプログラムを確認する

作成したプログラムの全体を [リスト 5.1](#) に示します。みなさんも作成したプログラムの全体を確認しましょう。

リスト 5.1 【Ruby】score.rb

```
2 # frozen_string_literal: true
3
4 require 'securerandom'
5
6 # Frameは1フレーム分のピン数やボーナスを記録する
7 class Frame
8   attr_reader :frame_no
9   attr_accessor :first, :second, :spare_bonus, :strike_bonus, :total, :state
10
11  def initialize(frame_no)
12    @frame_no = frame_no
13    @first = 0
14    @second = 0
15    @spare_bonus = 0
16    @strike_bonus = 0
17    @total = 0
18    @state = :RESERVED
19  end
20
21  def action(event, pins=0)
22    case @state
23    when :RESERVED
24      case event
25      when :SETUP
26        @state = :BEFORE_1ST
27      else
28        puts "invalid event: #{event} is ignored."
29      end
30    when :BEFORE_1ST
31      before_1st_proc(event, pins)
32    when :BEFORE_2ND
33      before_2nd_proc(event, pins)
```

```
34  when :PENDING
35      case evt
36      when :DETERMINE
37          @state = :FIXED
38      end
39      when :FIXED
40          puts 'fixed.'
41      end
42  end
43
44  def frame_score
45      @first + @second + @spare_bonus + @strike_bonus
46  end
47
48  def strike?
49      @first == 10
50  end
51
52  def spare?
53      @first < 10 && (@first + @second) == 10
54  end
55
56  def miss?
57      @first < 10 && @second.zero && @state == :FIXED
58  end
59
60  def gutter?
61      @first.zero
62  end
63
64  def fixed?
65      @state == :FIXED
66  end
67
68  def to_s
69      total = if @state == :FIXED
70          @total
71      else
72          ' . '
73      end
74      format '|%2d|%3s|%3s|%5s|%11d|%11d|%12d|%11s|',
75          @frame_no, @first, @second, total, frame_score,
76          @spare_bonus, @strike_bonus, @state
77  end
78
79  private
80
81  def before_1st_porc(evt, pins)
82      case evt
83      when :PINS
84          puts "invalid pins: #{pins}" if pins.negative? || pins > 10
85          @first = pins
86          @state = if strike?
87              @second = 0
```

```
88          :PENDING
89      else
90          :BEFORE_2ND
91      end
92  else
93      puts "invalid event: #{evt} on #{@state}.""
94  end
95 end
96
97 def before_2nd_proc(evt, pins)
98  case evt
99  when :PINS
100     puts "invalid pins: #{pins}" if pins.negative? || pins > (10 - @first)
101     @second = pins
102     @state = if spare?
103         :PENDING
104     else
105         :FIXED
106     end
107  else
108      puts "invalid event: #{evt} on #{@state}.""
109  end
110 end
111 end
112
113 # スコアは各人の10フレーム分のスコアを記録する
114 class Score
115   attr_accessor :id, :player, :fno, :frames, :state
116
117   def initialize(name)
118     @id = SecureRandom.urlsafe_base64(8)
119     @player = name
120     @fno = 1
121     @frames = []
122     (-1..13).each do |fno| # (-1, 0) are dummy frame
123       @frames.append Frame.new(fno)
124     end
125     @state = :WAIT_FOR_1ST
126     @frames[fno2idx(@fno)].action(:SETUP)
127   end
128
129   def fno2idx(fno)
130     fno + 1 # frame number 1 => array index 3 (0 origin).
131   end
132
133   def frame(fno)
134     @frames[fno2idx(fno)] # return index on @frames at frame number.
135   end
136
137   def go_next_frame
138     @fno += 1
139     @frames[fno2idx(fno)].action(:SETUP)
140   end
141
```

```
142 def current
143   frame(@fno)
144 end
145
146 def prev
147   frame(@fno - 1)
148 end
149
150 def pprev
151   frame(@fno - 2)
152 end
153
154 def calc_spare_bonus_after_1st
155   return unless prev.spare?
156
157   prev.strike_bonus = current.first
158   prev.action(:DETERMINE)
159 end
160
161 def calc_strike_bonus_after_1st
162   return unless prev.strike? && pprev.strike?
163
164   pprev.strike_bonus = prev.first + current.first
165   pprev.action(:DETERMINE)
166 end
167
168 def calc_strike_bonus_after_2nd
169   return unless prev.strike?
170
171   prev.strike_bonus = current.first + current.second
172   prev.action(:DETERMINE)
173 end
174
175 def update_total
176   @frames.each_cons(2) do |prev, cur|
177     cur.total = prev.total + cur.frame_score
178   end
179 end
180
181 def finished?
182   frame(10).fixed?
183 end
184
185 def wait_for_1st_proc(pins)
186   current.action(:PINS, pins)
187   calc_spare_bonus_after_1st
188   calc_strike_bonus_after_1st
189   update_total
190   if finished?
191     @state = :FINISHED
192   elsif current.strike?
193     @state = :WAIT_FOR_1ST
194     go_next_frame
195   else
```

```
196      @state = :WAIT_FOR_2ND
197    end
198  end
199
200  def wait_for_2nd_proc(pins)
201    current.action(:PINS, pins)
202    calc_strike_bonus_after_2nd
203    update_total
204    if finished?
205      @state = :FINISHED
206    else
207      @state = :WAIT_FOR_1ST
208      go_next_frame
209    end
210  end
211
212  def scoring(pins)
213    case @state
214    when :WAIT_FOR_1ST
215      wait_for_1st_proc(pins)
216    when :WAIT_FOR_2ND
217      wait_for_2nd_proc(pins)
218    when :FINISHED
219      puts 'finished'
220    end
221  end
222
223  def to_s
224    "Player:#{@player}, Score(id: #{@id}), Frame:#{@fno},
225 |No|1st|2nd|Total|Frame Score|Spare Bonus|Strike Bonus|Frame State|
226 #{@frames.join("\n")}"
227  end
228 end
229
230 # Gameクラスは複数名の1ゲーム分のスコアのセットを構成する
231 class Game
232   attr_reader :id, :turn, :scores
233
234   def initialize
235     @id = SecureRandom.urlsafe_base64(8)
236     @turn = 0
237     @scores = []
238   end
239
240   def entry(name = 'unknown')
241     @scores.append(Score.new(name))
242   end
243
244   def turn_player_name
245     @scores[@turn].player
246   end
247
248   def go_next_turn
249     @turn = (@turn + 1) % @scores.size
```

```
250 end
251
252 def playing(score_index, pins)
253   @scores[score_index].scoring(pins)
254   if @scores[score_index].fno > 10
255     go_next_turn if @scores[score_index].finished?
256   elsif @scores[score_index].current.state == :BEFORE_1ST
257     go_next_turn
258   end
259 end
260
261 def finished?
262   @scores.reject(&:finished?) == []
263 end
264
265 def to_s
266   "Game(id:#{@id}),\n#{@scores.join("\n")}"
267 end
268 end
269
270 # ScoreSheetは、複数名の複数回のGameを記録する
271 class ScoreSheet
272   attr_accessor :id, :time, :games
273
274   def initialize(date)
275     @id = SecureRandom.urlsafe_base64(8)
276     @play_date = date
277     @games = []
278   end
279
280   def add_game(games = 1)
281     games.times do
282       @games.append(Game.new)
283     end
284   end
285
286   def to_s
287     "Score Sheet Date: #{@play_date}(id:#{@id})"
288   end
289 end
```

5.1.2 テスト用のプログラムを追加する

作成したプログラムの末尾に、テストするためのプログラムを追加します（リスト 5.2）。実際にピン数を与えてスコアを作成しているのは「`game.playing`」の呼び出しだけです。このテストコードでは、ピン数のデータは、1ゲーム分のピン数をまとめています。しかし、スコアシートのプログラム本体はピン数を受けるたびに動作するように作られているので、ピン数を1投球ずつ与える方法でも動かせます。

リスト 5.2 【Ruby】score.rb(ファイル末尾に追加したテストコード部分)

```
292 if $PROGRAM_NAME == __FILE__ ①
293   sheet = ScoreSheet.new(Time.now) ②
294   sheet.add_game ③
295   game = sheet.games.last ④
296   game.entry('くぼあき') ⑤
297   game.entry('うえはら')
298   puts sheet ⑥
299
300   game_records = [ ⑦
301     # [6, 3, 9, 0, 0, 3, 8, 2, 7, 3, 10, 9, 1, 8, 0, 10, 10, 6, 4],
302     # [10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 5, 3],
303     [7, 0, 5, 5, 10, 10, 5, 4, 10, 7, 3, 5, 4, 7, 3, 7, 3, 4],
304     [6, 3, 9, 0, 0, 3, 8, 2, 7, 3, 10, 9, 1, 8, 0, 10, 6, 3]
305   ]
306
307 until game.finished? ⑧
308   puts '-----'
309   puts "turn: #{game.turn_player_name}"
310   score_index = game.turn ⑨
311   pins = game_records[score_index].shift ⑩
312   puts "input pins: #{pins}"
313   game.playing(score_index, pins) ⑪
314   puts game.to_s ⑫
315   gets ⑬
316 end
317
318 puts "Game(id:#{game.id}) is finished."
319 end
```

- ① 実行時プログラム名とファイル名が同じとき(このファイル自身から実行したとき)に実行する部分のはじまり。
- ② 新しいスコアシートを作成する。
- ③ 作成したスコアシートにゲームを追加する。
- ④ 追加したゲームを選ぶ(ここでは最後のゲームを選ぶ方法をとった)。
- ⑤ ゲームへプレーヤーがエントリーする。このプレーヤーのこのゲーム用のスコアが用意される。
- ⑥ スコアシートの作成日付とIDの表示。
- ⑦ テスト用データ。1行が1人の1ゲーム分のピン数のデータ。
- ⑧ ゲームの終了まで繰り返し。
- ⑨ turnは、そのゲーム内の何番目のプレーヤーか(ゲーム中の何番目のスコアか)を表している。このテストコードでは2人がエントリしているので、0→1→0→1と繰り返す。
- ⑩ 現在のturnのプレーヤーのピン数のテストデータを先頭から1つずつ取り出す。
- ⑪ 現在のturnのプレーヤーのスコアにピン数を渡して、スコアへ反映する。
- ⑫ 現在のゲームの全体像を取得して表示する。

- ⑯ ここで空の改行キーを入力してテストを進める。テストを1回ずつ止めて実行できる。この入力待ちをやめれば一括して実行できる。

5.2 プログラムをテストする

テスト用のプログラムも作成できたので、実行してみましょう。

コマンドプロンプトを起動して(MacやLinuxならターミナルを起動して)、プログラムを実行します(リスト5.3)。

出力フォーマットは、左から「フレーム番号」「1投目のピン数」「2投目のピン数」「のべのトータル」「フレームのトータル」「スペアボーナス」「ストライクボーナス」「フレームの状態」です。

フレームの状態の変化を追ってみて、フレームのステートマシン図と対応していることを確認してみましょう。

リスト5.3【端末】score.rbを実行する

```
C:\Users\kuboaki>cd Desktop\BowlingScore

C:\Users\kuboaki\Desktop\BowlingScore>ruby score.rb
Score Sheet Date: 2022-02-26 02:08:21 +0900(id:0GG70gNRWtk) ①
-----
turn: くぼあき ②
input pins: 7 ③
Game(id:zwBxlorJlog), ④
Player:くぼあき, Score(id: q5rZ-vnFM1o), Frame:1, ⑤
|No|1st|2nd|Total|Frame Score|Spare Bonus|Strike Bonus|Frame State|
|-1| 0| 0| .| 0| 0| 0| RESERVED| ⑥
| 0| 0| 0| .| 0| 0| 0| RESERVED|
| 1| 7| 0| .| 7| 0| 0| BEFORE_2ND| ⑦
| 2| 0| 0| .| 0| 0| 0| RESERVED|
| 3| 0| 0| .| 0| 0| 0| RESERVED|
| 4| 0| 0| .| 0| 0| 0| RESERVED|
| 5| 0| 0| .| 0| 0| 0| RESERVED|
| 6| 0| 0| .| 0| 0| 0| RESERVED|
| 7| 0| 0| .| 0| 0| 0| RESERVED|
| 8| 0| 0| .| 0| 0| 0| RESERVED|
| 9| 0| 0| .| 0| 0| 0| RESERVED|
| 10| 0| 0| .| 0| 0| 0| RESERVED|
| 11| 0| 0| .| 0| 0| 0| RESERVED|
| 12| 0| 0| .| 0| 0| 0| RESERVED|
| 13| 0| 0| .| 0| 0| 0| RESERVED|
Player:うえはら, Score(id: plq9pFD6nSA), Frame:1,
|No|1st|2nd|Total|Frame Score|Spare Bonus|Strike Bonus|Frame State|
|-1| 0| 0| .| 0| 0| 0| RESERVED|
| 0| 0| 0| .| 0| 0| 0| RESERVED|
| 1| 0| 0| .| 0| 0| 0| BEFORE_1ST|
| 2| 0| 0| .| 0| 0| 0| RESERVED|
| 3| 0| 0| .| 0| 0| 0| RESERVED|
```

4 0 0 . 0 0 0 RESERVED
5 0 0 . 0 0 0 RESERVED
6 0 0 . 0 0 0 RESERVED
7 0 0 . 0 0 0 RESERVED
8 0 0 . 0 0 0 RESERVED
9 0 0 . 0 0 0 RESERVED
10 0 0 . 0 0 0 RESERVED
11 0 0 . 0 0 0 RESERVED
12 0 0 . 0 0 0 RESERVED
13 0 0 . 0 0 0 RESERVED

(略)

turn: うえはら

input pins: 3

Game(id:zwBxlorJlog),

Player:くぼあき, Score(id: q5rZ-vnFM1o), Frame:11,

No	1st	2nd	Total	Frame Score	Spare Bonus	Strike Bonus	Frame State
-1	0	0	.	0	0	0	RESERVED
0	0	0	.	0	0	0	RESERVED
1	7	0	7	7	0	0	FIXED
2	5	5	27	20	10	0	FIXED
3	10	0	52	25	0	15	FIXED
4	10	0	71	19	0	9	FIXED
5	5	4	80	9	0	0	FIXED
6	10	0	100	20	0	10	FIXED
7	7	3	115	15	5	0	FIXED
8	5	4	124	9	0	0	FIXED
9	7	3	141	17	7	0	FIXED
10	7	3	155	14	4	0	FIXED
11	4	0	.	4	0	0	BEFORE_2ND ⑧
12	0	0	.	0	0	0	RESERVED
13	0	0	.	0	0	0	RESERVED

Player:うえはら, Score(id: plq9pFD6nSA), Frame:10,

No	1st	2nd	Total	Frame Score	Spare Bonus	Strike Bonus	Frame State
-1	0	0	.	0	0	0	RESERVED
0	0	0	.	0	0	0	RESERVED
1	6	3	9	9	0	0	FIXED
2	9	0	18	9	0	0	FIXED
3	0	3	21	3	0	0	FIXED
4	8	2	38	17	7	0	FIXED
5	7	3	58	20	10	0	FIXED
6	10	0	78	20	0	10	FIXED
7	9	1	96	18	8	0	FIXED
8	8	0	104	8	0	0	FIXED
9	10	0	123	19	0	9	FIXED
10	6	3	132	9	0	0	FIXED
11	0	0	.	0	0	0	RESERVED
12	0	0	.	0	0	0	RESERVED
13	0	0	.	0	0	0	RESERVED

Game(id:zwBxlorJlog) is finished.

C:\Users\kuboaki\Desktop\BowlingScore>

- ① スコアシートの作成日付とIDの表示。
- ② どのプレーヤーのターンなのかを表示した。
- ③ プログラムが受け取ったピン数。
- ④ 現在のゲームのID。
- ⑤ プレーヤーとそのプレーヤーのスコアのID、現在のフレーム番号。
- ⑥ 前の前のフレームを無条件に扱うためのダミーフレーム。
- ⑦ 1投目が終わったので、フレームは2投目の投球前の状態。
- ⑧ ここは2投目の投球前の状態だが、第10フレームの状態が「FIXED」なのでゲームは終了。

動作を確認できるよう、テストプログラムを実行した様子を収めた動画を用意しました。このテキストと一緒に配布している資料として `videos` フォルダに次のファイルを置いてあります。

実行の様子を収めたビデオ（`videos` フォルダにあります）

- `score_rb_demo_win.mp4`
- `score_rb_demo_mac.mov`

5.3 まとめ

作成したプログラムを、テストコードを使って動かしてみました。テストした結果、作成したプログラムが、モデル（クラス図やステートマシン図）に従って動作していることを確認できました。

6 まとめ

このチュートリアルでは、ボーリングスコアのモデルを作ることを通して、ソフトウェアの開発にモデルを使うことについて学びました。また、UMLが、そのようなときに役立つ記法であることも実感できたのではないでしょうか。ソフトウェアの開発にモデルを活用するときは、記法やツールも欠かせません。このチュートリアルでは、モデル図を描くのに astah* を使いました。ですが、このチュートリアルをやったみなさんは、記法やツールがあればモデルが作れるわけではないことに気づいたのではないでしょうか。

まず、このチュートリアルでは、ボーリングスコアの構造のモデルや振る舞いのモデルをいきなり作るのではなく、実際のスコアシートを観察してオブジェクト図を作るところから始めました。ボーリングスコアのような自明なよく知られた問題であれば、このような手間をかけなくてもモデルを作成できる人もいるでしょう。しかし、システム開発案件の多くは、どのように対処すればよいのかわかっていない問題を抱えています(だからこそ、そこが開発案件になるわけですよね)。このチュートリアルでは、そのようなときに問題を表すモデルを作るのと同じように、ボーリングスコアの記録に関わる領域をモデルで表すところから始めました。

みなさんのなかには、UMLの考え方や記法に慣れていないだけで、モデル化して考えることには慣れている人もいるでしょう。そのような人は、こんどは自分たちがすでにモデル化している図や文書を、UMLを使って表してみるとよいでしょう。

あるいは、すでに作りたいもの(あるいは解決したい課題)はわかってきているものの、それがどのような課題でどのように解決すればよいのかといったことを、うまく表せずに困っている人もいるでしょう。そのような人は、みなさんの課題(を含む現状)をモデル図で表してみるとよいでしょう。これを「As Is のモデル」と呼びます。そして、浮き彫りになった問題を、そのモデルを出発点として解消したモデルを作ります。これを「To Be のモデル」と呼びます。課題を解決したシステムは、To Be のモデルを元に設計、実装します。このような方法で、問題解決にモデルを活用してみることにチャレンジしてみてください。

みなさんがこれまで以上にモデルを活用できるようになることを期待しています。

7 他の図、他の機能など

この演習で作成したのは、クラス図(オブジェクト図も含む)とステートマシン図でした。UMLには他にも多くの図の記法が提供されています。ほかによく使われる図としては、ユースケース図、シーケンス図、アクティビティ図があります。

付録A: モデルやプログラムの作成例

ボウリングのゲームスコアのモデル図(`models` フォルダにあります)

- `bowling_score.asta`

Rubyで実装したプログラム(`codes` フォルダにあります)

- `score.rb`

実行している様子の動画(`videos` フォルダにあります)

- `score_rb_demo_win.mp4`
- `score_rb_demo_mac.mov`

参考文献

- [UMLSPEC] OMG 統一モデリング言語(UNIFIED MODELING LANGUAGE).
 - <https://www.omg.org/spec/UML/>
- [robustness01] ワークブック形式で学ぶUMLオブジェクトモデリング.
 - ローゼンバーグ, スコット. ソフトバンク. 2002.
- [robustness02] ユースケース駆動開発実践ガイド.
 - ローゼンバーグ, ステファン. 翔泳社. 2007.
- [rulebook] ボーリング競技規則.
 - <https://jbc-iwate.com/service/message/201012101.pdf>

モデルを使ってソフトウェアを開発しよう

UML初学者向けチュートリアル

発行日 : 2023-05-31

バージョン : pdf_0238

作成者 : 株式会社チェンジビジョン

本書の内容に関する質問等がありましたら、作成者までお知らせください。