

Design Assertions: Executable Assertions for Design Constraints

Yoonsik Cheon(✉)

The University of Texas at El Paso, El Paso, TX, USA
ycheon@utep.edu

Abstract. An assertion is a Boolean expression embedded in a program that must hold during the execution. Executable assertions are a simple but practical way to check assumptions and code logic at runtime. Assertions are written by referring to concrete program states. In this paper, we recognize a variety of assertions that we call *design assertions*. These are assertions written to ensure design constraints and properties, not detailed implementation decisions, and thus can detect major problems in the implementation such as design drift or corrosion. However, they are written by referring to concrete program states, thus causing readability and maintenance problems. To address these problems, we propose to write design constraints at a higher abstraction level by referring to abstract program states. We explain our approach using the Dart/Flutter platform, but it should work in other languages and platforms with similar assertion facilities.

Keywords: Assertion · Abstract model · Assertion-only method · Design constraint · Dart

1 Introduction

An assertion is a Boolean expression embedded at a specific place in a program that must hold during program execution. As evidenced by the presence of the *assert* statement and similar constructs in programming languages, assertions are recognized as a simple but practical programming tool. They provide a way to embed a programmer's assumptions and logic in source code in a form that can be checked at runtime. Assertions are written by referring to concrete program states, e.g., implementation-dependent data structures and algorithms. Therefore, when there are changes in implementation decisions, assertions have to be reformulated and rewritten. Furthermore, there are assertions to check design-level assumptions, constraints, or properties that should hold regardless of specific choices of implementation data structures and algorithms. In fact, assertions are said to be more effective when generated from design constraints and decisions. These assertions can detect a deviation or drift of implementation from its design, called *design corrosion* [14] or *decay* [11], one of the main problems associated with software maintenance.

In this paper, we study the use of assertions on the Flutter/Dart platform. Flutter is a cross-platform development framework for building apps that runs on multiple

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
O. Gervasi et al. (Eds.): ICCSA 2022 Workshops, LNCS 13381, pp. 617–631, 2022.
https://doi.org/10.1007/978-3-031-10548-7_45

platforms, e.g., Android and iOS [10]. Dart [4], the language of Flutter, provides an *assert* statement to write an executable assertion. In Flutter, a common pattern is to use assertions to perform preliminary checks on the constructor's arguments of a widget class. A Flutter widget is created and rendered automatically when its state changes. In Flutter, *assert* statements are enabled only in debug mode and are removed from production code in release mode. Flutter uses a dead code elimination technique called *tree shaking* to reduce compiled code size. This means that functions and classes used only in assertions are eliminated from the compiled code. Assertions, therefore, provide a simple yet effective way to check code logic in debug mode by stating the effects of a portion of code. We would like to promote the use of executable assertions for detecting bigger problems such as design drift or corrosion on the Dar/Flutter platform.

In this paper, we first show that there are two different kinds of executable assertions that we write in our code: *design assertions* and *code assertions*. Design assertions are assertions to check design constraints – design decisions, properties, and business rules – and must hold regardless of implementations. If expressed properly, they don't need to be reformulated or rewritten upon implementation changes such as data structures and algorithms. On the other hand, code assertions are for checking implementation-specific assumptions, properties, and decisions, e.g., choices of data structures and algorithms, and thus may need to be rewritten when there are changes in implementation decisions. We then propose an approach for writing design assertions abstractly in an implementation-independent way to make them more readable and maintainable. Two key elements of our approach are *assertion-only methods* and *abstract models*. Assertion-only methods are getters (special access methods in Dart) and observer methods whose sole purpose is to write assertions. We can also define a special assertion-only method called an *abstraction function* to map a concrete program state to an abstract model. The idea is to write assertions in terms of the abstract model rather than a concrete program representation. The assertion-only methods and the abstract models provide a vocabulary to write assertions at a higher level of abstraction without directly referring to concrete program states. Therefore, changes in implementation details have minimal or no impact on design assertions. Furthermore, this approach better supports some of the required properties of executable assertions such as transparency and plug-and-playability [6, 7].

An assertion-only method is similar to a specification-only method found in behavioral interface specification languages (BISL) such as JML [13]. It is a pure or side-effect-free method to be used only in assertions. An abstract model is similar to a model variable or a specification-only variable found in BISLs (e.g., [9]). It is a derived attribute or field to write assertions at a higher level of abstraction. Unlike in BISLs, we blend these specification concepts in the assertion facility of a programming language without extending the language or requiring special tool support. We enrich the vocabulary of Dart/Flutter assertions and make them more expressive. Our work was inspired by existing work on translating design constraints written in OCL [17] to runtime checks (e.g., [2]). The focus of our work, however, is the Dart/Flutter platform and its assertion facility. We think our approach is more practical and less holistic in that we suggest translating only key design constraints directly to executable assertions using the assert facility of the programming language. Thanks to the expressiveness of the language, most methods in well-written Dart/Flutter code are short and straightforward to the

extent that they don't need assertions. Assertions, for example, are frequently redundant and longer than the code itself. In previous work, the translation is done indirectly with the use of BISL such as JML [1, 12] or an aspect-oriented programming language such as AspectJ [5, 8, 15]. There is also a BISL proposed for Dart [6], but it doesn't include specification-only methods or abstract models.

The rest of this paper is organized as follows. In Sect. 2 below, we identify two different kinds of assertions that we call design and code assertions. We explain them with examples taken from a Flutter app that will be used as a running example throughout this paper. In Sect. 3, we first describe the problems of the conventional way of writing design assertions and then propose our approach for writing them at a higher level of abstraction – assertion-only methods and abstract models. In Sect. 4, we show more examples along with some interesting observations, findings, and discussions on writing design assertions. In Sect. 5, we conclude this paper with a concluding remark.

2 Design and Code Assertions

In this section, we identify two different kinds of assertions that we write in our code. We explain them by showing example assertions picked from a small app for playing connect-four games that we will use as a running example in this paper. Connect Four is a classic two-player connection game in which the players take turns to drop their discs, or tokens, from the top into a seven-column, six-row vertically suspended grid (see Fig. 1). The discs fall straight down, occupying the next available space within the column. The objective is to be the first to connect four of one's discs next to each other vertically, horizontally, or diagonally.

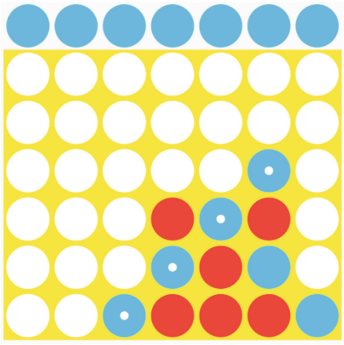

Fig. 1. Connect-four game

The class diagram in Fig. 2 shows one possible design for the model part of the connect-four app. A board representing the grid consists of a collection of places representing the spaces in the grid. Each place may be associated with one of the two players. The association indicates that a place is taken by a player and thus has a disc of the player. There are several different ways to realize the Board class in an implementation. We can use different data structures to store all the places of a board: a two-dimensional

array-like structure (an array of columns or an array of rows), a one-dimensional array (stored in column/row/arbitrary order), a map from indexes to places, etc.

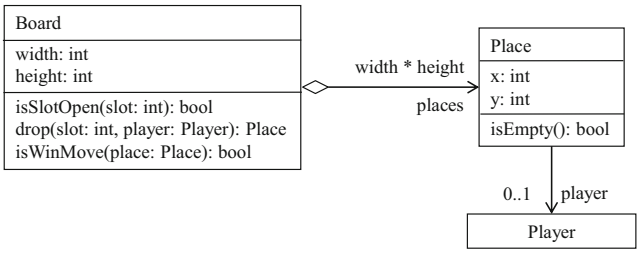

Fig. 2. Connect-four class diagram

The following code snippet of the Board constructor shows one possible implementation choice. All places are stored in a list (*_places*) in column-major order; in Dart, a name starting with an underscore (_) is private. The code also shows three *assert* statements, the Dart way of writing executable assertions. The first statement checks the parameters. The *assert* statement is very commonly used to check assumptions or the precondition of a method or constructor. For example, it is a common pattern for Flutter widget classes to perform preliminary checks on the constructor's parameters. The last two statements check code logic – postconditions or facts on the state that the code has to establish.

```
Board(int width, int height) {
  assert(width >= 7 && height >= 6);
  this.width = width;
  this.height = height;
  _places = List.generate(width * height,
        (i) => Place(i ~/ height, i % height), growable: false);
  assert(_places.length == width * height);
  assert(_places.every((p) =>
    _places.indexOf(p) == p.x * height + p.y));
}
```

The above three assertions have fundamental differences in their natures. The first assertion is about a design-level assumption, i.e., the dimension of a board or a connect-four grid. On the other hand, the other two are concerned with an implementation-level decision, i.e., a particular choice of data structure. A list is used to store all the places on a board, and thus its length is constrained. Similarly, the indexes of elements are constrained because the elements are stored in column-major order. The assertions may need to be reformulated and rewritten if the implementation decision changes. In this paper, we will use the terms *design assertions* and *code assertions* to refer to these two kinds of assertions. Design assertions are assertions to check design constraints – design decisions, properties, and business rules – and must hold regardless of implementations. If expressed properly, they don't need to be reformulated or rewritten when there are

implementation changes such as data structures and algorithms. On the other hand, the purpose of code assertions is to check implementation-specific decisions, e.g., choices of data structures and algorithms, and thus may need to be rewritten upon changes in implementation decisions.

Below we show another code snippet with assertions. One key operation of the Board class is to place a player's disc in an empty place. The *drop()* method shown below implements this operation. Given a 0-based column index and a player, the method drops a disc of the player in the column. The *for* loop finds the last empty place in the column and associate it with the player.

```
Place drop(int slot, Player player) {
  assert(0 <= slot && slot < width);
  for (var i = (slot + 1)  * height -1; i >= slot * height; i--) {
    if (_places[i].isEmpty) {
      _places[i].player = player;
      assert(_places.getRange(slot * height, i)
          .every((p) => p.isEmpty));
      assert(_places.getRange(i, (slot + 1) * height)
          .every((p) => p.isFilled));
      return _places[i].;
    }
  }
  assert(_places.getRange(slot * height, (slot + 1) * height)
    .every((p) => p.isFilled));
  return null;
}
```

The code has three *assert* statements, and their usage pattern is very similar to the previous example. The first assertion is the precondition of the method and checks the validity of the parameters. The other three check the effect of the code in the post-state just before the control returns to the caller. The two nested *assert* statements together state that a player's disc is placed in the bottom-most empty place of a column. The last statement asserts that if a disc isn't placed, the column has no empty place. The way the three assertions are written gives the impression that they are code assertions. The assertions are formulated in terms of implementation-specific features such as the index of the *_places* list. Are they really code assertions? They assert (partial) behavior of the operation that should hold regardless of implementations; it is an operation postcondition expressed in terms of an implementation-specific data structure. In fact, it asserts partially a key invariant of the Board class: each column of a board is filled from the bottom to the top.

```
assert(Iterable.generate(width).every((slot) =>
   _places.getRange(slot * height, (slot + 1) * height )
      .every((p) => p.isEmpty ? true :
         _places.getRange(_places.indexOf(p) + 1,
           (slot + 1) * height)
            .every((q) => q.isFilled))));
```

3 Writing Design Assertions

3.1 The Problem

As shown in the previous section, assertions are written by referring to concrete program states – variables, fields, and methods. For code assertions, it may be acceptable because they have to be rewritten anyway if the program representation changes. It is often desirable to express the required relationships among elements of the program state, e.g., the *x* and *y* coordinates of a place and the position of the place in the containing list. For design constraints, however, there are several problems associated with expressing them in terms of the concrete program state. One problem is that when the representation, e.g., data structure, changes, assertions have to be reformulated and rewritten. As an example, consider changing the representation of the Board class. Instead of storing all the places in a one-dimensional list, we can organize them as a list of columns. This change requires us to review all occurrences of the *_places* field in the code including the assertions. For example, we need to rewrite the last three assert statements of the *drop*() methods as follows.

```
for (var i = height -1; i >= 0; i--) {
  if (_places[slot][i].isEmpty) {
    _places[slot][i].player = player;
    assert(_places[slot].getRange(0, i)
      .every((p) => p.isEmpty));
    assert(_places[slot].getRange(i, height)
      .every((p) => p.isFilled));
    return _places[slot][i].;
  }
}
assert(_places[slot].getRange(0, height)
  .every((p) => p.isFilled));
```

The required changes may be small or the rewritings may be done systematically. However, the concern is that all design assertions have to be rewritten manually, causing maintenance issues. Another problem is that when assertions are written by referring to concrete program states, it is not clear or easy to tell whether they are code assertions or design assertions. It is said that assertions are more effective when they are generated from design constraints and properties. These assertions called design assertions can detect major issues or problems compared with those of code assertions. They can detect when an implementation deviates from its design, often called *design corrosion* [14] or *design decay* [11], one of the main problems associated with software development and maintenance. We will need to pay more attention when design assertions are violated. Ideally, design assertions should be written in terms of the design vocabulary – design elements and concepts – not the implementation vocabulary. In the following two subsections, we propose two different ways to solve the above problems by writing design assertions abstractly.

3.2 Algebraic-Style Assertions

We can eliminate or minimize direct references to concrete program states by using getters or observer methods. In Dart, getters are special methods that provide read access to an object's properties or fields. For example, instead of accessing a place in a list directly, e.g., *_places*[*slot * height + y*], we can use an observer method, say *_getPlace*(*int slot*, *int y*). If there is no such method already defined, we can introduce one just for the purpose of writing assertions. The getters and observer methods allow us to access the concrete program states indirectly. Therefore, if we change the program representation, i.e., data structure, we need to make changes only to the getter and observer methods, thus minimizing the impact of the change to the assertions. The assertions also become more readable, as they are written in the vocabulary of a higher level of abstraction. In a sense, we enrich the vocabulary of the assertion language by providing model- or design-level operations. As an example, let us rewrite the *drop*() method of the Board class (see Sect. 2). The code snippet below shows the rewritten method. We introduce a new observer method *_getColumn*(*int*) that, given a slot index, returns a column of places. With this method, we can eliminate the use of the *_places* field from the two assertions in the middle. For the last one, we can also replace the whole assertion with a call to the *isSlotFull*() method that checks whether a given slot is filled completely.

```
Place drop(int slot, Player player) {
  assert(0 <= slot && slot < width);
  for (var i = (slot + 1)  * height - 1; i >= slot * height; i--) {
    if (_places[i].isEmpty) {
      _places[i].player = player;
      assert(_getColumn(slot).getRange(0, i - slot * height)
        .every((p) => p.isEmpty));
      assert(_getColumn(slot).getRange(i - slot * height, height)
        .every((p) => p.isFilled));
      return _places[i];
        }
      }
      assert(isSlotFull(slot));
      return null;
    }

    @assert_only
    List<Place> _getColumn(int slot) =>
      _places.getRange(slot * height, (slot + 1) * height);
```

When introducing getters and observer methods only for assertions, it would be a good idea to indicate it. We can define an annotation for this, say *@assert_only*, as shown above. The annotation also means that the annotated method should have no observable side-effect. An assertion expression may have side effects for the purpose of assertion checking, but the side-effect shouldn't be observable by the implementation code [7]. An assertion-only method should not be used in the implementation code. On the Flutter platform, assertions are enabled only in debug mode and are completely ignored in

release mode in that they are not evaluated. Flutter also uses a dead code elimination technique called *tree shaking* to reduce code size. Functions and classes that are never used are removed from the compiled code. Therefore, assertion-only methods are not compiled in the production code in release mode.

As shown in the above example, we can write more abstract assertions by accessing the concrete program representation indirectly through getters and observer methods. If needed, we can also introduce assertion-only getters and observers, thus enriching the vocabulary for writing assertions. The set of getters and observers used in assertions defines implicitly an abstract view of a concrete program state. This style of writing assertions is in a sense algebraic. An operation is constrained by other operations – getters and observers. In other words, an assertion specifies a relationship or constraint among operations. Therefore, one fundamental question of this approach is: how do we write assertions for the getters and observer methods themselves? Ultimately, we will need to refer to the concrete program states. This algebraic style assertions may require a large number of assertion-only methods and may be less intuitive to a programmer because assertions are not constructive.

3.3 Abstract Model

The previous approach defines an abstract model of a program implicitly by the set of getters and observer methods used in assertions. We can define an abstract model explicitly and use it in assertions. An abstract model is an assertion's view on a concrete program state and provides the vocabulary to write assertions (see Fig. 3). Since an abstract model will be used only in assertions, it should be immutable or no mutation method should be allowed. By manipulating an abstract model in assertions, we may be able to write assertions in a constructive fashion, which may be more intuitive than an algebraic style to most programmers (see Sect. 4).

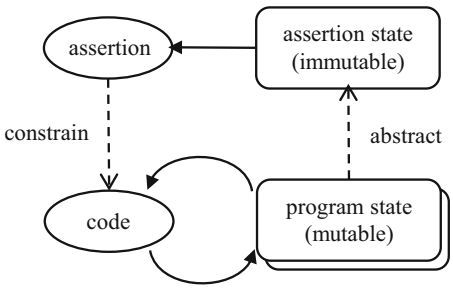

Fig. 3. Abstract model approach

In our implementation of the Board class, all places of a board are stored in a one-dimensional list in column-major order for performance reasons. Assertions, however, are enabled only in debug mode and stripped off from production code in release mode. And we are not or less concerned about performance of assertions, so, it would be advantageous to view all places as a list of columns. We can define explicitly a mapping,

called an *abstraction function*, from a concrete program representation to an abstract representation or model. For example, we can define a private getter named *_model* as shown follows.

```
@assert_only
List<List<Place>> get _model => List.unmodifiable(
   Iterable.generate(width).map((slot) => List.unmodifiable(
     _places.getRange(slot * height, (slot + 1) * height))));
```

The getter returns an abstraction of the current program state, i.e., the *_places* field. It returns an immutable list to avoid side effects in assertions. If needed, we can also introduce assertion-only classes to express an abstract model just like we do with assertion-only methods (see Sect. 4). Once we define an abstraction function, we can write assertions to check various design constraints and properties in terms of the abstract model. For example, the following two assertions check class invariants saying that (a) a board has *width* * *height* number of places, and (b) the *x* and *y* indexes of each place are unique and are in the range of 0 and *width* - 1 and 0 and *height* - 1, respectively. They can be added to the constructors to ensure the establishment of the invariants and to the mutation methods to ensure the preservation [7].

```
assert(_model.length == width &&
   _model.every((column) => column.length = height));
assert(Iterable.generate(width).every((x) =>
   Iterable.generate(height).every((y) =>
        _model[x][y].x == x && _model[x][y].y ==y));
```

Of course, we can use the abstract model to rewrite assertions for methods and get rid of direct references to a concrete program state (see below).

```
Place drop(int slot, Player player) {
 assert(0 <= slot && slot < _model.length);
 for (var i = (slot + 1)  * height -1; i >= slot * height; i--) {
  if (_places[i].isEmpty) {
    _places[i].player = player;
    assert(_model[slot].every((p) =>
      p.y < _places[i].y ? p.isEmpty : p.isFilled));
    return _places[i];
  }
 }
 assert(_model[slot].every((p) => p.isFilled));
 return null;
}
```

Two most common types of design constraints are class invariants and operation pre and postconditions. The ways these constraints are written are different. An invariant is a constraint on a single state and thus needs to refer to only a single abstract model.

It only needs to observe the values in a single state, not the changes of the values. On the other hand, a postcondition is frequently a constraint on two states, pre- and post-states, to specify the side effect of an operation, and documents state changes or transformations by referring to two abstract models. It needs to constrain the changes to the state, i.e., from the pre-state to the post-state. We can make the whole pre-state or some parts available to the post-state *assert* statements by introducing (assertion-only) local variables.

```
Place drop(int slot, Player player) {
  assert(0 <= slot && slot < _model.length);
  final y_ = _model[slot].lastIndexWhere((p) => p.isEmpty);
  for (var i = (slot + 1)  * height -1; i >= slot * height; i--) {
    if (_places[i].isEmpty) {
      _places[i].player = player;
      assert(y_ >= 0);
      assert(_model[slot][y_].player == player);
      return _places[i]; // assert(places[i] == _model[slot][_y]);
    }
  }
  assert(y_ < 0);
  return null;
}
```

The local variable *y_* was introduced to store a pre-state value of an expression. It is initialized with the last row index of an empty place contained in the column under consideration. The three post-state assertions can be written in terms of this assertion-only local variable. The use of assertion-only local variables improves the expressiveness of assertions and thus lets us write more rich assertions. We can assert not only the side effects of a section of code but also when the side effects should happen. The above three assertions, for example, state that when there is an empty place in the given slot ($y_ >= 0$), the place is filled; otherwise ($y_ < 0$), no change in the state assuming an appropriate frame axiom such as "and nothing else changes" [3]. One downside of introducing a local variable for the sole purpose of writing assertions is that it may produce an "unused variable" warning in a release build because all *assert* statements are removed.

The abstract model approach can be thought of as a restricted form of the algebraic style approach where one assertion-only method defines an abstract model and the others, if any, are defined in terms of this abstract model. And design assertions are written in terms of this model without directly referring to the concrete program states. Therefore, one clear advantage is that when there are changes in the program representation, only a single assertion-only method, i.e., the abstraction function, needs to be reformulated and rewritten. By making the model immutable, we can also ensure the side-effect freeness of assertions. Another benefit is that we can tell easily which assertions are sort of more important design-level constraints and which are code-level constraints. However, one shortcoming is that an assertion violation is reported in terms of an abstract model, and thus we need to map it to a concrete program state to track and find the cause. In summary, the key idea of the abstract model approach is to define a model at a higher

level of abstraction to write design assertions with the vocabulary of design concepts and elements.

4 More Examples, Observations, and Discussions

Another key operation of the Board class is to determine whether there is a winning sequence of discs, four discs of the same player next to each other vertically, horizontally, or diagonally. The *isWinMove()* method below implements this operation. Given a place (of the last drop), it checks whether there exists a winning sequence containing the place. If a winning sequence exists, the method also has the side effect of storing the sequence in a field named *winRow*.

```
bool isWinMove(Place place) {
  assert(place.isFilled);
  var result = false;
  // … omitted code to find a winning sequence
  // assertion-only nested functions: isWin and seqs
  isWin(seq) => seq.fold(0, (cnt, p) => cnt >= 4 ? cnt :
    (p.player == place.player ? cnt + 1 : 0)) >= 4;
  seqs(places) => [
    places.where((p) => p.x == place.x),
    places.where((p) => p.y == place.y),
    places.where((p) => p.x - place.x == p.y - place.y),
    places.where((p) => p.x - place.x == -(p.y - place.y))];
  assert(result == seqs(_model.expand((col) => col))
     .where((seq) => isWin(seq)).isNotEmpty);
  assert(result ? winRow.length == 4 && isWin(winRow)
     : winRow == null);
  return result;
}
```

The above code shows that we can also introduce assertion-only nested functions. The *isWin*() function tests if a given sequence of places contains a winning sequence, and the *seqs*() function finds the row, column, and two diagonals containing the place in question. The last two assertions constrain the return value and the side effect of the method, respectively.

As said previously, invariants are an important type of design constraint that can be checked with assertions. More interesting invariants are those that involve multiple objects and constrain relationships among them. Below we show several assertions for class invariants to show some interesting observations and findings. We have a class named ViewModel to manage the app state. It stores data that is shared by multiple widgets. When there is a change in the app state, all widgets are notified, rebuilt, and rendered again.

```
class ViewModel extends Model {
  final Board board;
  bool gameOver;
  final List<Player> _players;
  Player _currentPlayer;
  // … other fields and methods
}
```

Below we have two invariants involving players. The first invariant states that there are two distinct players, and the current player, the player who has the turn, is one of the two players. The second invariant states that for every place on the board, its player should be one of the two players.

```
bool _inv1() => _players.contains(_currentPlayer)
   && _players.length == 2 && _players[0] != _players[1];

bool _inv2() =>  board.places.every((p) =>
      p.isEmpty || _players.contains(p.player));
```

The two invariants are similar in that we can write them by using the vocabulary of the implementation code, i.e., getters and observers. However, there is a big difference in their natures. The objects involved in the first invariant, the two players and the current player, are owned and managed by the ViewModel class, and the ViewModel class is responsible for ensuring the invariant. Therefore, we can call the invariant method in the *assert* statements embedded in the constructors and mutation methods such as *changeTurn*() of the ViewModel class. The invariant should be established by the constructors and preserved by the mutation methods of the ViewModel class. On the other hand, the places of the board referenced in the second invariant are not owned or managed by the ViewModel class; they are owned by the Board class. Therefore, the second invariant has to be checked by the Board class or its component class Place – i.e., when a place changes its state. The problem is the assertion can't be written inside the Board class because the two players are not available there and there is no reference from the Board class to the ViewModel class. One finding is that the right class in which to write a class invariant may not be the right class to check the invariant. It is future work to address this problem. A possible intrusive workaround would be to pass the assertion (or callback) to the Board class, e.g., through an assertion-only parameter or setter method.

There is also an invariant that is concerned with the *gameOver* field. The field can be thought of as a derived attribute in that its value can be calculated from other elements of the class. It is true when the board is full or contains a winning sequence. The following shows an incorrectly written assertion. The problem is that the *isWinMove*() method has a side effect and thus shouldn't be used in an assertion. And the Board class doesn't provide its client with a vocabulary suitable for writing the assertion.

```
bool _inv3() => gameOver == (board.isFull()
   || board.places.indexWhere((p) =>
      p.isFilled && board.isWinMove(p)) >= 0);
```

The real cause of the above problem is that when we write assertions outside a class, we have a limited vocabulary, i.e., public getters and observer methods of the class, as we can't access directly the hidden representation of the class. This vocabulary, a subset of the public API of the class, is for writing client code, and thus may not be suitable for writing client assertions. This raises an interesting discussion. Should a class provide assertion-only public features such as getters and observer methods for its clients? Would it be a good idea to have an assertion-only API for a class? For the above invariant, for example, we can introduce an assertion-only observer, say *hasWin*(), or make the abstract model public. Making the abstract model public seems to be the most general solution, and this is another strength of the abstract model approach.

We can write assertions in two different styles: *constrained* and *constructive styles*. In the constrained style, we constrain the value of a variable by specifying the properties that it has to satisfy without expressing it explicitly. In the constructive style, we specify the value explicitly by calculating it. We can use both styles to assert the return value of an operation or the state change that an operation has to establish. As an example, consider the *changeTurn*() method that alternates the two players' turns. We can assert its side-effect in the two different styles as follows.

```
bool changeTurn() {
  final oldCurrentPlayer_ = _currentPlayer;
  …
  assert(_currentPlayer != oldCurrentPlayer_);  // constrained
  assert(_currentPlayer ==                      // constructive
    _players.firstWhere((p) => p != oldCurrentPlayer_);
}
```

The first assertion written in the constrained style states that the new value of the *_currentPlayer* field should not be the same as its previous value. As there are only two players, the new value should be the other player. The second assertion is written in the constructive style in that it calculates the new value of the *_currentPlayer* field. It selects a player from the players list that is not the same as the pre-value of the *_currentPlayer* field. A constructive assertion is written in the form of $x == E$, where x is a variable and E is one specific value. Which style is preferred? It depends on the assertion, but our preference is the style that can express the intention of the assertion clearly and concisely. For the above assertion, we prefer the constrained style.

We found that with a suitable representation of a class, most methods are short and straightforward to the extent that they don't need assertions or assertions are redundant. In fact, oftentimes, assertions are longer than the code.

```
bool isFull() => _places.every((p) => p.isFilled);

bool isFull() { // rewritten to add an assertion
  var result = _places.every((p) => p.isFilled;
  assert(result ==
    _model.expand((col) => col).every((p) => p.isFilled));
  return result;
}
```

Most assertions shown in this paper are concerned with collections of values, e.g., a list of places and a list of two players. To write these assertions, we used higher-order methods defined on collections such as *every*(), *expand*(), *firstWhere*(), *fold*(), *indexWhere*(), and *where*(). These higher-order methods are indispensable for writing assertions involving a collection, but they are all read-only operations for accessing or observing the elements of a collection iteratively. In assertions, we can't use write operations that have side effects, e.g., adding new elements, removing existing elements, and performing an action on each element. However, to manipulate a collection of values in various ways, we need such operations but without side effects, e.g., an *add*() operation that returns a new collection. We, therefore, think that a mathematical toolkit or immutable collection library found in formal specification languages such as Z [16] and OCL [17] would be also useful for writing executable assertions. Such a library would be particularly helpful for defining abstract models as well as writing design assertions in a constructive style.

5 Conclusion

We presented our approach for writing executable assertions to check design constraints such as class invariants at runtime. The key idea of our approach is to write these assertions called *design assertions* at a higher level of abstraction without directly referring to the concrete program state. For this, we can introduce so-called *assertion-only* methods – getters and observer methods whose sole purpose is to write assertions. We can also define a getter called an *abstraction function* to map a concrete program state to an abstract model, and then we can write assertions in terms of the abstract model rather than a concrete program representation. The assertion-only methods and the abstract model provide a vocabulary to write assertions at a higher level of abstraction. Our approach allows programmers to embed runtime checks of important design decisions in code in a more readable and maintainable fashion. Changes in the implementation-level decisions such as data structures require no or minimal rewriting of design assertions; e.g., only the abstract function needs to be redefined. The two key contributions of our work include (a) recognition of an important class of assertions that we call design assertions, and (b) ways to write them at a higher level of abstraction. We also made several interesting observations, findings, and discussion topics, e.g., constrained- vs. constructive-style assertions, checking invariant assertions, and assertion-only APIs.

References

1. Avila, C., Flores, G., Cheon, Y.: A library-based approach to translating OCL constraints to JML assertions for runtime checking. In: International Conference on Software Engineering Research and Practice, Las Vegas, Nevada, 14–17 July, pp. 403–408 (2008)
2. Avila, C., Sarcar, A., Cheon, Y., Yeep, C.: Runtime constraint checking approaches for OCL, a critical comparison, In: International Conference on Software Engineering and Knowledge Engineering, 1–3 July, San Francisco, CA, pp. 393–398 (2010)
3. Borgida, A., Mylopoulos, J., Reiter, R.: '. . . and nothing else changes': the frame problem in procedure specifications. In: Proceedings of 1993 15th International Conference on Software Engineering, pp. 303–314. IEEE (1993)
4. Bracha, G.: The Dart Programming Language. Addison-Wesley, Boston (2016)
5. Briand, L., Dzidek, W., Labiche, Y.: Instrumenting contracts with aspect-oriented programming to increase observability and support debugging. In: International Conference on Software Maintenance, Budapest, Hungary, 25–30 September, pp. 687–690. IEEE (2005)
6. Chalin, P.: Ensuring that your dart will hit the mark: an introduction to dart contracts. In: International Conference on Information Reuse and Integration, Redwood City, CA, 13–15 August, pp. 369–377. IEEE (2014)
7. Cheon, Y.: Toward more effective use of assertions for mobile app development. In: International Conference on Progress in Informatics and Computing, Shanghai, China, 17–19 December, pp. 319–323. IEEE (2021)
8. Cheon, Y., Avila, C., Roach, S., Munoz, C.: Checking design constraints at run-time using OCL and AspectJ. Int. J. Softw. Eng. **2**(3), 5–28 (2009)
9. Cheon, Y., Leavens, T., Sitaraman, M., Edwards, S.: Model variables: cleanly supporting abstraction in design by contract. Softw. Pract. Exp. **35**(6), 583–599 (2005)
10. Flutter Homepage. https://flutter.dev/. Accessed 6 Nov 2021
11. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
12. Hamie, A.: Translating the object constraint language into the java modeling language. In: ACM Symposium on Applied Computing, Nicosia, Cyprus, 14–17 March, pp. 1531–1535. ACM (2004)
13. Leavens, G., Cheon, Y., Clifton, C., Ruby, C., Cok, D.: How the design of JML accommodates both runtime assertion checking and formal verification. Sci. Comput. Program. **55**(1–3), 185–208 (2005)
14. Perry, D., Wolf, A.: Foundations for the study of software architecture. ACM SIGSOFT Softw. Eng. Notes **17**(4), 40–52 (1992)
15. Richters, M., Gogolla, M.: Aspect-oriented monitoring of UML and OCL constraints. In: AOSD Modeling with UML Workshop, San Francisco, CA, 20 October 2003 (2003)
16. Spivey, S.: The Z Notation: A Reference Manual. Prentice Hall, Hoboken (1992)
17. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA, 2nd edn. Addison-Wesley, Hoboken (2003)