A Survey of Assert Forms in Io Embedded Unit Tests

Ward Cunningham

Draft of Feb 26, 2005

This report classifies the various forms of unit tests present in a single application developed using an experimental unit-testing framework for test-driven development. We find that there are 17 different kinds of asserts that fall into 6 broad categories. We consider this diverse for a 1000 line application and consider it a confirmation of the utility of embedded unit tests.

An embedded unit test is one that exists in the source code of the program being tested. Unlike traditional asserts, which also exist within the code, embedded unit tests inject test data into the program and expect that test data to be evaluated during the compilation process. An embedded unit testing framework must protect the finished program from damaged during testing.

Our experimental unit testing framework is implemented in the prototype based object oriented programming language Io. (Reference http://iolanguage.com) Io is a good host for our experiment because, 1) it evaluates expressions as it compiles, 2) it optimizes cloning of objects, 3) the program can control the context within which expressions are evaluated, and, 4) a readable print representation can be extracted from unevaluated expressions. We use all four capabilities when we define the modest lookiing unit testing interface:

> *anObject* assert (*testExpression*, *expectedResultExpression*)

This assert clones anObject (for isolation), evaluates testExpression (which may have multiple statements with side effects) in the context of the clone, and compares the result to the evaluated expectedResultExpression after it is evaluated in the context of the caller. It is our belief that this is sufficent capability to effectively develop correct and flexible code by the test-driven method. The full text of the unit testing framework is included in Appendix A.

The rest of this paper lists unit tests extracted from the multi-user interactive game, IoGame (Reference http://c2.com/~ward/io/IoGame). In this program asserts were written in small groups. We've preserved this grouping here except when more than one style test was written in the group. In the few occasions where this happened, we have separated the tests and replicated the fragments of context.

We gave a name (gray italic) to each resulting group that describes as simply as possible the nature of the test performed. These named groups were then reordered so that the simpler tests would be first and more general categories (afinity groups) would be obvious. We named these larger groups (orange), added text that explained them, and then explained each of the test groups after they had found there proper place. The completed list of categorized tests appears in Appendix B.

Appendix A

This is the definition of assert used in this experiment. The writeln is optional and has been included here because Io gives writeln more powerful arbitrary object formatting than available to the exception mechanism.

```
assert := method (
    actual := self clone doMessage (thisMessage argAt(0))
    expected := sender doMessage (thisMessage argAt(1))
    if (expected != actual,
        writeln (
            self type .. ": ",
             thisMessage argAt(0) code,
             " ==> ", actual, " not ", expected
        )
         Error raise ("Assert Failed", self type .. ": " .. thisMessage argAt(0) code)
```

```
        )
    )
```

Appendix B

Global Scope

These tests retrieve objects from the global namespace.

Navigate

Can we retrieve elements from the list of suits and get the ones we expect? We test this by examining the slots of what we get. The correct order for colors was taken from the rainbow.

```
red := Suit clone do (name = "red"; color = "ffcccc"; id = 0)
yellow := Suit clone do (name = "yellow"; color = "ffff88"; id = 1)
green := Suit clone do (name = "green"; color =  "ccffaa"; id = 2)
blue := Suit clone do (name = "blue"; color =  "bbddff"; id = 3)

suits := list (red, yellow, green, blue)

assert (suits at (3) color, "bbddff")
assert (suits at (1) name, "yellow")
assert (suits at (green id), green)
```

Equality

Do the various suits implement equality correctly, especially the wild suit which is equal to all others. Io represents true with anything not nil. Many operators return something useful but here I prefered to not know how truth was represented. I think a one argument version of assert that implicitly checked for truth would avoid this problem while remaining consistent with other languages with simpler asserts. (The unit testing tradition of taking multiple arguments comes from wanting to report more meaningful errors on assert failures.)

```
wild := WildSuit
wild id = 5

assert ((red == red) isNil , Nil)
assert ((red == green), Nil)
assert ((wild == yellow) isNil, Nil)
assert ((yellow == wild) isNil, Nil)
```

Constructor

Here we teach number how to make playing cards. This is a convenience syntax to simplify entering of examples. Again we test by examining slots of the objects produced.

```
Number do (
        red := method (new := PlayingCard clone; new suit = Lobby red; new rank = self; new)
        yellow := method (new := PlayingCard clone; new suit = Lobby yellow; new rank = self; new)
        green := method (new := PlayingCard clone; new suit = Lobby green; new rank = self; new)
        blue := method (new := PlayingCard clone; new suit = Lobby blue; new rank = self; new)
)

assert ((1 green) type, "PlayingCard")
assert ((3 yellow) rank, 3)
assert ((5 red) suit, red)
```

Mutator

We found a need to convert examples to another suit. Here we construct a list of cards, mutate their suits, then pick one out and

examine it in more detail. Note that even though this test is evaluated in the global context, the functional nature of the mutator (changeAll) means that the program remains unadulterated by the tests.

```
Suit := Object clone do (
	type := "Suit"
	name := Nil
	id := Nil
	color := "ffcccc"
	isWild := Nil
	setSlot ("==",
		 method (aSuit,
			(name == aSuit name) or aSuit isWild
		)
	)
	changeAll := method (aCardList,
		aCardList foreach (i, card,
			card suit := self
		)
		aCardList
	)
)

assert (blue changeAll (list (1 red, 2 green)) at (1) suit, blue)
assert (wild changeAll (list (1 red, 2 green)) at (1) suit, wild)
```

Ignored Proto Scope

In Io the do(...) method evaluates a series of statements in the context of the receiver. Often this is used to simplify the setting up of a series of slots in a prototype. However, here the convention of clustering statements within a do(...) provides only organizational value.

Literal Setup

Here we add some capability to String and test it on a literal.

```
String do (
	html := method (
		self replace (" ", "<br>") replace ("_", " ")
	)
	assert ("four of_a kind" html, "four<br>of a<br>kind")
)
```

Used Instance Scope

Io enforces no distinction between prototypes and instances. However, by convention, we say that an object that exists primarilly to hold runtime state is an instance and is given a lower case name while an object that primarilly exists to organize behavior that is inherited by cloning is called a prototype and given an upper case name.

The assert command evaluates its left argument in the context of the receiver. (Actually, a clone. See below.) These are tests where the receiver was an algorithmetically constructed instance that could benefit from testing. We actually had some confusion with the indexing and intruduced assert to unravel it.

Indexing

Here we just check that cards are given ids that match their initial position in the deck. This fact is only true before the first goal card is drawn, but we check it so that we can believe that all goal cards will be consecutivly numbered.

```
goalDeck do (
	addCardForEachSuit (FlushGoal clone do (cards = 3; goal = "three card flush"))
	addCardForEachSuit (FlushGoal clone do (cards = 4; goal = "four card flush"))
	assert (at (0) id, 0)
	assert (at (1) id, 1)
```

```
	assert (at (7) id, 7)
)
```

Indexing

We generate a lot more playing cards and have refactored this code several times. The later tests exploit asString that didn't exist when the first tests were written.

```
playingDeck := Deck clone do (
	nextId := 0
	addNewCards := method (
		list (red, yellow, green, blue, wild) foreach (i, suit,
			newSuit := PlayingCard clone
			newSuit suit = suit
			list (0,1,2,3,4,5,6,7,8,9) foreach (j, rank,
				new := newSuit clone
				new id = nextId; nextId = nextId + 1
				new assert (rank, rank)
				add (new)
			)
		)
	)

	addNewCards
	assert (at(0) id, 0)
	assert (at(1) id, 1)
	assert (at(0) asString, "0 red")
	assert (at(10) asString, "0 yellow")
	assert (at(11) asString, "1 yellow")
)
```

Indexing (poor)

These asserts are not written in what I would consider good style. That is because instance that is the focus of our testing has been brought inside the arguments of the assert. There is an important difference between these two cases:

1. *assert (ourGoals at(4) type, "GoalCard")*
2. *ourGoals assert (at(4) type, "GoalCard")*

The instance, ourGoals, is protected from side-effect in the second form because the receiver, ourGoals, is cloned before the argument, at(4) type, is evaluated. The first form runs without this protection so if the argument were to have side-effects it would leave ourGoals side-effected before the program finishes compiling. The indexing examples above are using the second, safer form but with the receiver provided by the do(...) message.

```
ourGoals := Goals clone do (
	add (goalDeck draw)
	add (goalDeck draw)
	add (goalDeck draw)
	add (goalDeck draw)
	add (goalDeck draw)
)

assert (ourGoals at(4) type, "GoalCard")
assert (ourGoals detect (i, card, card id == ourGoals at(2) id) id, ourGoals at(2) id)
assert (ourGoals position (ourGoals at(2) id), 2)
assert (ourGoals at(3) type, "GoalCard")
assert (ourGoals at(3) id type, "Number")
```

Modified Proto Scope

These tests exploit the implict clone of the receiver prformed by every assert. Each of these tests modify the the clone in some way and then test that these modifications had the desired effect. The effects don't last, though. The clone is discarded at the end of the assert.

Method Setup

We modify the built-in abstraction for time intervals to format using a less precise but more user friendly notation. We test this method by setting different interval magnitudes into the clones of the receiver (the prototype, Duration) and then trying the conversions. The first argument of the assert is everything up to the comma, in this case, two statements separated by a semicolon.

```
Duration do (
	asString := method (
		s := (asNumber floor)
		if ((m := ((s / 60) floor)) < 2, return s asString .. " seconds")
		if ((h := ((m / 60) floor)) < 2, return m asString .. " minutes")
		if ((d := ((h / 24) floor)) < 2, return h asString .. " hours")
		if ((w := ((d /  7) floor)) < 2, return d asString .. " days")
		if ((m := ((d / 30) floor)) < 2, return w asString .. " weeks")
		if ((y := ((d /365) floor)) < 2, return m asString .. " months")
		y asString .. " years"
	)
	assert (setSeconds (5); asString, "5 seconds")
	assert (setSeconds (90); asString, "90 seconds")
	assert (setSeconds (120); asString, "2 minutes")
	assert (setDays (14); asString, "2 weeks")
	assert (setDays (20); asString, "2 weeks")
)
```

Categorization

The SampleHand prototype offers methods for sorting cards in ways that are convenient when checking to see if a goal is satisfied. Here we test those sorts by adding some cards to the implicit SampleHand clone and then looking through the sorted versions.

```
SampleHand := CardList clone do (
	bySuit := method (
		cardsBySuit := List clone
		suits foreach (i, suit,
			cardsBySuit add (self select (j, card, card suit == suit))
		)
		cardsBySuit // with wilds duplicated
	)
	byRank := method (
		cardsByRank := list (Nil,Nil,Nil,Nil,Nil,Nil,Nil,Nil,Nil,Nil)
		self foreach (i, card,
			rankOfcard := card rank
			cardsOfRank := cardsByRank at (rankOfcard)
			if (cardsOfRank,
				cardsOfRank add (card),
				cardsByRank atPut (rankOfcard, list (card))
			)
		)
		cardsByRank
	)
	assert (add (4 red); add (6 green); count, 2)
	assert (add (4 red); bySuit at(0) count, 1)
	assert (add (3 green); byRank at(3) count, 1)
	assert (add (3 green); add (3 yellow) byRank at(3) count, 2)
)
```

Manipulation

We manage Decks of cards with two lists, one representing the draw pile, the other the discard pile. These tests verify that we can't get a card from an empty deck, but that when we add one card (to the draw or the discard pile) that card becomes available. Note: the implicit clone protects the discard pile by making a new one with each clone in an "init" method. This good programming practice would be required if the methodology promoted here were to be used to its fullest.

```
Deck := List clone do (
	type := "Deck"
	discards := List clone
```

```
	init := method (
		self discards := List clone
	)
	draw := method (
		if (first isNil,
			shuffleDiscards
		)
		card := self random
		remove (card)
		card
	)
	discard := method (card,
		discards add (card)
	)
	shuffleDiscards := method (
		addList (discards)
		discards empty
	)

	assert (draw, Nil)
	assert (add (Card clone do (id = 5)); draw id, 5)
	assert (discard (Card clone do (id = 6)); draw id, 6)
)
```

Calculation

A goal card is worth points. The exact number of points can be calculated from other quantities and using other methods, as in this case where a 'resend()' invokes the inherited implementation of points. This assert sets up a sample situation and then checks the delegation and subsequent adjustment.

```
AnySuitFlushGoal := FlushGoal clone do (
	suit := AnyOneSuit
	points := method ((resend () / 2) floor)
	assert (cards = 5; points, 12)
)

goalDeck do (
	addCard (AnySuitFlushGoal clone do (cards = 5; goal = "five card flush"))
	addCard (AnySuitFlushGoal clone do (cards = 4; goal = "four card flush"))
)
```

Translation

We use buffer objects to accumulate html. Here we test that a properly initiialized buffere is in fact equal to the equivalent string. Another assert tests an html generating macro, in this case one that draws the bar of a bar chart.

```
Markup := Buffer clone do (

	html := method (string, self append (string))
	assert (html("foo"), "foo")

	bar := method (average,
		width := ((average * 100) floor)
		html ("<br><br><table bgcolor=#dddddd height=20 width=" .. width .. "><tr><td> " .. average .. " </table>")
	)
	assert (bar (2.23456), "<br><br><table bgcolor=#dddddd height=20 width=223><tr><td> 2.23456 </table>")
)
```

Simulated Context

These tests create more complex situations, including the simulation of data sources and sinks.

Exhaustion

The game can be played with any number of players which raises the question of what happens when one runs out of cards.

We wrote the logic to essentially create additional decks on demand and tested it by dealing an unreasonable number of cards. This code has never been tested in practice because we have yet to assemble ten simultainous players, the number required to exhaust the first deck.

```
playingDeck := Deck clone do (
	nextId := 0
	addNewCards := method (
		list (red, yellow, green, blue, wild) foreach (i, suit,
			newSuit := PlayingCard clone
			newSuit suit = suit
			list (0,1,2,3,4,5,6,7,8,9) foreach (j, rank,
				new := newSuit clone
				new id = nextId; nextId = nextId + 1
				new assert (rank, rank)
				add (new)
			)
		)
	)
	shuffleDiscards := method (
		if (discards first,
			resend,
			addNewCards
		)
	)

	addNewCards
	assert (101 repeatTimes (draw); nextId, 150)
)
```

Interpretability

The look and feel of the game was tested by exploratory testing (Refference). However, this test was added just to be sure each of the major screens could be generated without error. A sentinal value (done) is added to the testExpression to simplify result checking. The presence of random numbers in the algorithms makes more thorough testing overly difficult. At one time these tests used the distinguished Io object Nop (no operation) as a universal mock of the markup. This proved unnecessary because markup is sufficiently robust to operate without error in this faux environment. Note: these are the only unit tests who's setup spills outside of the assert line itself.

```
Turn := Object clone do (
	type = "Turn"
	query := Nil
	hand := Nil
	markup := Nil

	commands := list ("about", "play", "players", "docs", "community", "discard", "satisfy", "explain", "trace")
	doCommand := method (number, self doString (commands at(number)))
)


Turn clone do (
	hand = Hand clone
	markup = Markup clone
	assert (about; play; players; docs; community; "done", "done")
	assert (query = hand first id; discard; "done", "done")
	assert (query = ourGoals first id; explain; "done", "done")
	hand throwIn
	assert (playingDeck count + (playingDeck discards count), 50)
)
```

Initialization Method

Integrity

The goal and playing decks are constructed algorithmically. Here we check that each goal added to the deck does not have an

id before we give it one, and that the goal judges its sample had playable. This one assert exercises 20% of the game.

```
goalDeck := Deck clone
goalDeck addCard := method (card,
        card assert (id, Nil)
        card id = goalDeck count
        add (card)
        card assert ((isPlayable (sampleHand)) isNil, Nil)
)
```

Implicit

A late refactoring was to rely on the orderly assigned id numbers to serve double duty as the source of card rank also. That is, card rank could be inferred from id by modulo arithmetic. Here the loop that assigns id checks that this id leads to a rank equal to that that would have been assigned in an earlier version of the method. This is the most dramatic demonstration of the distinct evaluation contexts provided for the two arguments to assert.

```
playingDeck := Deck clone do (
        nextId := 0
        addNewCards := method (
                list (red, yellow, green, blue, wild) foreach (i, suit,
                        newSuit := PlayingCard clone
                        newSuit suit = suit
                        list (0,1,2,3,4,5,6,7,8,9) foreach (j, rank,
                                new := newSuit clone
                                new id = nextId; nextId = nextId + 1
                                new assert (rank, rank)
                                add (new)
                        )
                )
        )
)
```