Metaphor-Validity Survey

A quick summary of what this survey is all about: we are developing a set of metaphors to aid novice programmers in establishing high-fidelity mental models of fundamental programming concepts - the metaphors created are intended to be usable across several different media including educational games, textbooks, and classroom environments. The metaphors presented here have been through several rounds of internal testing, followed by several rounds of user testing with students in order to identify and rectify potential problem areas.

At this stage we are of the opinion that further testing through students will no longer yield meaningful results as they cannot generally compare the metaphors being presented to them with their own mental models of programming concepts (either because they don’t any or because they are inaccurate or incomplete), which is where this survey comes in: We hope to draw on the knowledge and existing mental models held by experienced programmers in order to further analyse, improve on, and validate the metaphor set we have come up with.

So as to avoid ‘contaminating’ test subject opinions, the questions for each concept-metaphor pair are broken down into two sections: pre-questions about how the subject sees and imagines a concept, and post-questions which focus more on the metaphors we are proposing.

The final questions for each section are perceived scores:

* Understandability - How easy is it to grasp what the metaphor is trying to explain, and how it works?
  + 1 - I couldn't understand it at all.
  + 5 - I had to think about it but I got it in the end.
  + 10 - Perfectly intuitive and easy to grasp.
* Versatility - How widely used could the metaphor be, or how many concepts do you think it could explain?
  + 1 - I don't think it relates to the concept in question ever.
  + 5 - I could use it to explain about half the scenarios I can think of.
  + 10 - It works everywhere so far as I can see.
* Durability - How likely is this metaphor to fail or break down?
  + 1 - This metaphor is so inaccurate it breaks down right out the gate.
  + 5 - It won't break down for simple example.
  + 10 - This metaphor seems as if it wouldn't fail in even the most complex situations.
* Relatability - How well does the given metaphor relate to your own mental model or understanding of a concept?
  + 1 - It doesn't relate at all.
  + 5 - It sort of relates but not across the board.
  + 10 - It's almost like you read my mind.
* Accuracy - How well does the given metaphor compare to the way a particular concept, structure, or function actually works?
  + 1 - The two aren't remotely similar.
  + 5 - The metaphor is a reasonable representation some of the time.
  + 10 - The metaphor is a perfect representation of what happens behind the scenes.

Two quick pre-pre-questions:

Q. Approximately how many years of programming do you have?

A.

Q. Please rank your favourite programming languages from most to least favourite:

A.

Values in a Programming Language

Pre-Questions

Q. If you were to explain values to a student as a real-world metaphor, how would you do it?

A.

Q. Do you have a mental model for values, if so could you try to explain it?

A.

Q. Can you think of any scenario where your mental model or proposed metaphor for values becomes inaccurate?

A.

Our Metaphor

The most fundamental things that code uses are individual values, either as actual values or the value of a reference address (this would include integers, doubles, Booleans, chars, and all other primitives). As this is such a key concept, its associated metaphor needs to be one of the most reliable and easy to understand: it was proposed that a simple piece of paper with the value written on it would be suitable. Everyone can relate to pen and paper, you cannot erase pen from paper (meaning values are never `re-used'), and if one were to imagine solving an expression in their head the most sensible thing to do with the answer would be to write it down. Figure 1 shows an example of a value notepad, the user inputting a value, and finally the user holding the value.

|  |
| --- |
|  |
| Figure 1. A: An int value-notepad. B: User inputting a value. C: The user holding the final value. D: Alternative value-notepads. |

Post-Questions

Q. How does the metaphor we give for values compare with how you imagine it working?

A.

Q. Can you find fault with our metaphor for values, if so what would it be?

A.

Q. If you could alter one thing about this metaphor what would it be?

A.

Q. Please score our proposal -

A. Understandability \_\_\_%, Versatility \_\_\_%, Durability \_\_\_%, Relatability \_\_\_%, Accuracy \_\_\_%

Variables in a Programming Language

Pre-Questions

Q. If you were to explain variables to a student as a real-world metaphor, how would you do it?

A.

Q. Do you have a mental model for variables, if so could you try to explain it?

A.

Q. Can you think of any scenario where your mental model or proposed metaphor for variables becomes inaccurate?

A.

Our Metaphor

Variables are represented by boxes, with transparent lids, that contain a single piece of paper (in the same way that a simple variable can contain a single value). Reading of a variable would be done courtesy of the transparent lid; you would simply look into the variable box and copy the value off the paper without changing the content of the box. Assigning to a variable involves opening the box, disposing of the old value-paper, and then placing the new value (which you would be holding) into the box. Figure 2 shows one potential version of the variable box metaphor. The representation of variables on the stack comes later - up to this point it is enough for users to know that variables exist

and how to imagine them.

|  |
| --- |
|  |
| Figure 2. A variable box called **Counter**, containing the integer value 12345678. |

Post-Questions

Q. How does the metaphor we give for variables compare with how you imagine it working?

A.

Q. Can you find fault with our metaphor for variables, if so what would it be?

A.

Q. If you could alter one thing about this metaphor what would it be?

A.

Q. Please score our proposal A. Understandability \_\_\_%, Versatility \_\_\_%, Durability \_\_\_%, Relatability \_\_\_%, Accuracy \_\_\_%

Expressions, Arithmetic and Calculation

Pre-Questions

Q. If you were to explain expressions and calculation to a student as a real-world metaphor, how would you do it?

A.

Q. Do you have a mental model for expressions and calculation, if so could you try to explain it?

A.

Q. Can you think of any scenario where your mental model or proposed metaphor for expressions and calculation becomes inaccurate?

A.

Our Metaphor

Expressions of all kinds have two viable metaphors in our unified set: they can either go through an in-game calculator, or the user can work them out in their head. Good arguments can be made for either case, and thus both metaphors have been included so that anyone who wants to expand on this work can make up their own mind. The original technique for expressions and evaluation was the use of an in-game calculator, which worked much like a real-world scientific calculator, where users could enter an expression and then go back and alter terms once they had the required values. For example a user might enter “x + 5” into their calculator, they would then look at the local variables and copy the value from x onto a piece of value paper, that value would be fed into the calculator (almost like a fax machine, except the paper is destroyed).

When the user asks to substitute into the place holder `x' in the expression, the calculator would replace the variable in the expression and then wait for further instructions from the user (either for more instructions or for the user to ask it to evaluate the answer). When the expression no longer has variables that need values, the user would hit `evaluate' and the answer would be printed out from the calculator onto a piece of paper. Figure 3 illustrates the sequence of events when using the calculator metaphor.

An alternative technique for expression evaluation actually takes away extra metaphors entirely by just asking the user to evaluate the expression in their heads (or on paper, or with their real-world calculator), in much the same way as when users debug a piece of code. Steps A, B and C in Figure 1 demonstrate how users would enter the answer to an expression.

|  |
| --- |
|  |
| Figure 3. The steps required to evaluate an expression using the in-game calculator. A: the calculator is blank and requires an expression. B: An expression has been entered and now requires value-substitutions for every variable. C: A value has been substituted (this requires a variable read first), this step is repeated 3 times. D: All values have been inserted and the user has asked for the answer (which was printed onto a piece of value-paper). |

Post-Questions

Q. How does the metaphor we give for expressions and calculation compare with how you imagine it working?

A.

Q. Can you find fault with our metaphor for expressions and calculation, if so what would it be?

A.

Q. If you could alter one thing about this metaphor what would it be?

A.

Q. Please score our proposal

A. Understandability \_\_\_%, Versatility \_\_\_%, Durability \_\_\_%, Relatability \_\_\_%, Accuracy \_\_\_%

Local Variables and the Current Stack Frame

Pre-Questions

Q. If you were to explain the current stack frame (perhaps think local scope) to a student as a real-world metaphor, how would you do it?

A.

Q. Do you have a mental model for the current stack frame, if so could you try to explain it?

A.

Q. Can you think of any scenario where your mental model or proposed metaphor for the current stack frame becomes inaccurate?

A.

Our metaphor

The next step up the abstraction ladder is to delimit scope and accessibility using some kind of uncrossable barrier (for example a line of barbed wire, police tape, or even a simple fence - Figure 4 shows two possible barriers). This barrier serves to separate the current frame (i.e. the local scope) from the other stack frames (as well as global variables from heap variables). The non-local area, and everything in it, will be discussed later; scope-delimitation was mentioned here for clarification of the next concept to be described: how to represent the current frame and all the variables that are stored within it, while still being relatable, and without complicating the pushing of stack frames onto the stack.

|  |
| --- |
|  |
| Figure 4. Top: Barbed wire memory barrier. Bottom: Police tape memory barrier. |

We represent the current frame as something akin to a jigsaw or Lego bookshelf: users would start out without a bookshelf at all when there are no local variables, a mechanism which dispenses bookshelf `pieces' is all that would be present. Whenever the user needs to declare a new variable they would first have to extend the bookshelf so there would be enough space for it. Each space on the bookshelf would be able to contain one variable box. The expandable bookshelf metaphor allows you to teach students about how local variables can go out of scope, for example when they were declared inside a loop, the out-of-scope variables (and their associated bookshelf sections) can be removed entirely. The metaphorical bookshelf sits on top of a conveyor belt, this only becomes important when the user is able to call methods, thus it is discussed in more detail in the next section. Figure 5 shows what the user might be presented with, depending on the current stack frame state.

|  |
| --- |
|  |
| Figure 5. The current frame and local variables bookshelf - displayed on top of the frame conveyor belt, and directly accessible. Demonstrating the various states during the declaration of local variables `a' to `g'. |

Post-Questions

Q. How does the metaphor we give for the current stack frame and local scope compare with how you imagine it working?

A.

Q. Can you find fault with our metaphor for the current stack frame and local scope, if so what would it be?

A.

Q. If you could alter one thing about this metaphor what would it be?

A.

Q. Please score our proposal

A. Understandability \_\_\_%, Versatility \_\_\_%, Durability \_\_\_%, Relatability \_\_\_%, Accuracy \_\_\_%

The Stack, Without Method Functionality

Pre-Questions

Q. If you were to explain the whole stack (rather than just one frame) to a student as a real-world metaphor, how would you do it?

A.

Q. Do you have a mental model for the stack, if so could you try to explain it?

A.

Q. Can you think of any scenario where your mental model or proposed metaphor for the stack becomes inaccurate?

A.

Our metaphor

Once one understands how to interpret the representation of the current frame, one is then also able to interpret the stack as a whole: just like a library usually has more than one bookshelf, a stack usually has more than one frame - therefore we can cross the two ideas and represent the stack as rows of bookshelves. This is also where the accessibility divide becomes important, as the stack is primarily located of the directly accessible space. When a new frame is created from a method call, all non-local stack frame bookshelves are moved across the memory divisor so that they exist in memory space. Figure 6 shows how the stack would expand into memory space after each method call, and shrink after each return statement. For clarity the memory space includes the heap, the non-local stack frames, and nothing else.

Post-Questions

Q. How does the metaphor we give for the stack compare with how you imagine it working?

A.

Q. Can you find fault with our metaphor for the stack, if so what would it be?

A.

Q. If you could alter one thing about this metaphor what would it be?

A.

Q. Please score our proposal

A. Understandability \_\_\_%, Versatility \_\_\_%, Durability \_\_\_%, Relatability \_\_\_%, Accuracy \_\_\_%

|  |
| --- |
|  |
| Figure 6: Changes in state and visualisation for the stack and the frames that correspond to each method call. Read from left to right this shows the calling of each method, while read from right to left it demonstrates returning back down the stack (pushing and popping frames respectively). |

Methods and Method Functionality

Pre-Questions

Q. If you were to explain methods and their associated functionality to a student as one or more real-world metaphors, how would you do it?

A.

Q. Do you have a mental model (or more than one) for all things method-related, if so could you try to explain it (or them)?

A.

Q. Can you think of any scenario where your mental model/s or proposed metaphor/s for methods becomes inaccurate?

A.

Our metaphor

Know that we know what the stack looks like we can examine the metaphors governing everything to do with methods.

As mentioned previously the current frame's associated bookshelf is positioned on top of a conveyor belt, this allows bookshelves to be moved across the barbed wire and into or out of thefix all. This mechanism is important for method calls and returns: the conveyor belt has two control buttons, the call button, and the return button. These two buttons move everything on top of the conveyor belt either toward or away from the directly accessible space. So far it is not hard to imagine the moving of the conveyor belt as matching up with the actual process of pushing and popping frames to and from the stack; the tricky part of designing this section of the metaphor set is considering what happens to frames that have been popped off the stack, and where new frames come from. Both situations can be explained in a similar way to overwritten variable values: For popped frames and overwritten values the object in question no longer belongs anywhere and so must be destroyed, the paper value would be torn up or burnt, and a similar thing would be done to the popped off frame. In the same way that we prepare paper values using notepads we decided to use a workbench to represent the creation of a soon to be used frame.

After getting the student to name the method they are preparing to call, you have several options for method preparation that you can present them with. The two proposed techniques are as follows: the first way is to ask users to name and assign parameters and arguments in much the same way as local variables (one at a time, with types, name, and values all the explicit responsibility of the user); the second possible presentation method is to give users a selection of available method signatures which they need to pick from, and then automatically provide all the parameters ready to receive values, which the user simply needs to fill. Figure 7 compares the two main method preparation techniques side-by-side.

Much like the two possible ways of asking students to evaluate expressions, these two method mechanisms are both valid and the technique used will depend on the instructor or implementation. For a void method no further explanation is really required for how the user leaves the method (they simply press the conveyor belt return button). However it might need saying that a value returning method works by simply making the user hold the return value in their hand before pressing the return button, that way the return value is in hand when they get back to the previous frame, and thus they can use the value straight away.

Post-Questions

Q. How do the metaphors we give for methods compare with how you imagine them working?

A.

Q. Can you find fault with our metaphors for methods, if so what would it be?

A.

Q. If you could alter one thing about this metaphor what would it be?

A.

Q. Of the two possible representations, which do you prefer and why?

A.

Q. Initial usability tests have shown that students pick up the signature-picking metaphor far faster than all the other metaphors explained so far, do you have any opinions on why that might be?

A.

Q. Please score our proposal

A. Understandability \_\_\_%, Versatility \_\_\_%, Durability \_\_\_%, Relatability \_\_\_%, Accuracy \_\_\_%

|  |
| --- |
|  |
| Figure 7. Left: Metaphor one for method preparation, where the user is responsible for everything. Right: metaphor two (signature-picking metaphor) for method preparation, where the user is responsible for arguments and calling but nothing more. The stages from top to bottom are: no method named, a method named but no parameters assigned or declared, one parameter (a) named and assigned, all the parameters named and assigned with the method ready for calling. |

Global Variables

Pre-Questions

Q. If you were to explain global variables to a student as one or more real-world metaphors, how would you do it?

A.

Q. Do you have a mental model (or more than one) for global variables, if so could you try to explain it (or them)?

A.

Q. Can you think of any scenario where your mental model/s or proposed metaphor/s global variables becomes inaccurate?

A.

Our Metaphor

More modern programming languages don’t strictly speaking have truly global variables anymore, instead we have class fields. So in this context global variables refer to class fields of whatever context we are currently in.

Firstly, because global variables are always accessible, they need a representation inside the user accessible space. Secondly, because they exist almost from the very start of the program the user needn’t be able to create more of them (unlike local variables). These two facts can be brought together with another bookshelf metaphor: a fixed size bookshelf that has all the necessary variables already declared. Aesthetically one can represent this sort of bookshelf in more than one way, our proposed representations is shown in Figure 8. If the number of globals is awkward, one could instead use the jigsaw representation used for the local stack frame. A potential concern for some might be that globals should be created explicitly much like locals, this point is debatable.

|  |  |
| --- | --- |
| D:\Dropbox\My Work\Masters Work\Documents\Images\globalsDemo.jpg | Figure 8. A simple fixed-size bookshelf that holds the global variables. It is directly accessible to the user. |

Post-Questions

Q. How do the metaphors we give for global variables compare with how you imagine them working?

A.

Q. Can you find fault with our metaphors for global variables, if so what would it be?

A.

Q. If you could alter one thing about this metaphor what would it be?

A.

Q. Please score our proposal

A. Understandability \_\_\_%, Versatility \_\_\_%, Durability \_\_\_%, Relatability \_\_\_%, Accuracy \_\_\_%

Representing the Heap, Objects and Instances (but not communicating with either just yet)

Pre-Questions

Q. If you were to explain the heap and objects (or reference types) to a student as one or more real-world metaphors, how would you do it?

A.

Q. Do you have a mental model (or more than one) for the heap and objects, if so could you try to explain it (or them)?

A.

Q. Can you think of any scenario where your mental model/s or proposed metaphor/s for the heap and objects becomes inaccurate?

A.

Our metaphor

Reference types in the local scope can easily be represented by a simple memory address written down as a value. The more challenging thing to represent about reference types is what gets stored in memory (outside of immediately addressable space). Without any additional metaphors there is no way for players to affect what exists outside of the directly accessible space, to solve this issue we introduce a robot to represent the memory manager, which is elaborated on later. For now it is sufficient to say that the memory manager robot receives messages from the user, and carries them out in memory space.

With our memory manager robot ready to interact with the heap and non-local stack on our behalf, we now need a way to represent the heap and access to the non-current frames on the stack. Access to the stack is fairly simple as we already have a concrete metaphor for the stack itself (the rows of bookshelves): the memory manager robot simply takes the address he has been given and goes between the bookshelves to interact with the appropriate variable box.

The heap requires more thought. After considering that the size of the heap is in fact finite (determined by the available memory) in an actual computer, we decided that the heap metaphor could also be represented by a finite, fixed-size structure. The easiest way to represent it without deviating from our existing metaphors is to have one super-sized bookshelf, possibly with movable dividers, where unused space is represented by the lack of a variable box. Figure 9 shows an example of this extra-large bookshelf, and also illustrates that in certain media the bookshelf representation falls slightly short due to the size limitations. For this reason, in-game, the heap can instead be shown as a compact series of squares (as shown in Figure 10), this alternative representation can just be thought of as a compact version of the bookshelf.

Representing objects, that are more complex than arrays of values, is where reference types become particularly difficult to represent: A simple non-array object such as a Random generator could be represented on the heap as two numbers and something to point to the objects associated method code. For this example the numbers would be the start seed and the current seed; however the methods belonging to the object would have to be seen only in the code that the students are following. Figure 11 shows a potential representation of the heap with instances of Random and Account classes, unlike Figure 9 the objects on this heap have their properties labelled so as to make it clear to users that they are not just arrays of numbers. The `name' property of the Account class would be a string, its contents on the heap would depend on how one chooses to represent strings,

as explained in the final section of the survey.

|  |
| --- |
|  |
| Figure 9. The heap bookshelf with two objects in it. If they were array objects they would be of length 4 and 6 respectively. This heap can only contain 36 simple values, and thus demonstrates that we might need a more compressed representation. |

|  |
| --- |
|  |
| Figure 10. A compressed representation of the heap. A: an empty heap. B and C: one array present on the heap, with details above. D: highlighting the free space. E: Three arrays on the heap, with ys being highlighted. G: a demonstration of what might happen after a garbage collection when ys is no longer being used. |
|  |
| Figure 11. A small section of the heap bookshelf showing potential representations of two  non-array objects. |

Post-Questions

Q. How do the metaphors we give for the heap and objects compare with how you imagine them working?

A.

Q. Can you find fault with our metaphors for the heap and objects, if so what would it be?

A.

Q. If you could alter one thing about this metaphor what would it be?

A.

Q. Please score our proposal

A. Understandability \_\_\_%, Versatility \_\_\_%, Durability \_\_\_%, Relatability \_\_\_%, Accuracy \_\_\_%

Interacting with objects and other reference types

Pre-Questions

Q. If you were to explain object referencing and communication with the heap to a student as one or more real-world metaphors, how would you do it?

A.

Q. Do you have a mental model (or more than one) for object referencing and communication with the heap, if so could you try and explain it (or them)?

A.

Q. Can you think of any scenario where your mental model/s or proposed metaphor/s for object referencing and communication with the heap becomes inaccurate?

Our metaphor

A.

Once you understand how the heap and various reference types are represented, communication with them becomes fairly simple to understand as well. The memory manager robot is responsible for all interactions in the memory space, he receives his instruction via the terminal (which is located in directly accessible space). When writing to the heap, values go from user to terminal to robot to memory address (and the other way for reading from the heap). In order to access data on the heap the user needs to provide the memory manager robot with two things: the objects base address, and the offset of the specific object-element of interest. For non-array objects the `offset' would instead be the name of the field or property in question. Reference access to variables on the stack is done in much the same way, except there is no need to provide an offset, an address alone is sufficient.

In order to facilitate easier state recognition the memory manager robot can have several ways of showing whether it currently has an address, offset or value: either lights on its torso can show up saying what it has (as shown in Figure 12), or it could hold visible values. These two metaphors don't require much explaining as they are just middlemen for communication with the metaphors of importance.

|  |
| --- |
|  |
| Figure 12. The memory manager robot (right) and the terminal used to communicate with him (left). This version of memory space communication was geared towards array only communication. |

Post-Questions

Q. How does the metaphor we give for object referencing and communication with the heap compare with how you imagine it working?

A.

Q. Can you find fault with our metaphor for object referencing and communication with the heap, if so what would it be?

A.

Q. If you could alter one thing about this metaphor what would it be?

A.

Q. Please score our proposal

A. Understandability \_\_\_%, Versatility \_\_\_%, Durability \_\_\_%, Relatability \_\_\_%, Accuracy \_\_\_%

Strings

Pre-Questions

Q. If you could represent strings as either objects or value types (bearing in mind the trade-offs between usability and accuracy), which would you choose and why.

A.

Our metaphor

Strings deserve a special mention regarding their representation; this is because while they are strictly speaking objects, their immutability means that treating them as value types is unlikely to cause problems. For this reason it was proposed that strings could have two representations in our metaphor set, depending on the preference of the user or teacher: one can either use the more accurate (but more bulky) option of treating them as objects on the heap, or they can be treated as value types that reside on the stack.

If one chooses to represent strings as objects on the heap then they would be treated as arrays of characters, with the special property of being read only after their initial declaration (due to their immutability). When they are treated as value types the metaphors don't break down: when you assign one to another you can treat them as though they are copies rather than aliases. As briefly mentioned when explaining the heap and objects, objects on the heap that have string properties would have different representations based on the choice of string representation: if we use the more accurate model with strings as objects then string properties would contain memory addresses, and the strings themselves would exist elsewhere on the heap as seemingly separate objects. If one uses the strings-as-value representation then string properties become simple, with the strings clearly being inside the parent object.

Post-Questions

Q. Given our explanation of strings as either objects or value types, would you change your previous answer, why/why not?

A.