Lego MindStorms as a Training Tool for Software Development Methodology in Multimedia Education.

Elias Wyber
Murdoch University
South St, Murdoch
WA, 6150, Austrailia
+61 (0) 411 422 070
e.wyber@murdoch.edu.au

Abstract
This paper outlines an approach to educating multimedia practitioners utilizing Lego MindStorms® which evolved out of a desire to bridge the technical and aesthetic worldviews. The approach focuses on three aspects of the development process. First, it examines the linguistic model of code production. In order to convey the act of writing code as a linguistic task, the students are introduced to several aspects of linguistic theory, with examples illustrating their application in spoken, programmed and Lego MindStorms® based languages. The second area of focus is object theory. Adopting a similar approach to the previous section, but giving only Lego based examples. Finally, the focus shifts to prototyping, discussing how Lego MindStorms® can be used to nurture an iterative development style. The conclusion then discusses the results of this approach, as it was applied at the Hypermedia Research Centre (HRC) in the University of Westminster between 1999 and 2002.

Keywords
Multimedia, Education, Linguistic, Programming, Lego, MindStorms, Design, Theory.

1. Introduction
Historically, new media developers tend to be from one of two backgrounds, either technical (particularly in the case of backend and HTML/JavaScript programmers) or arts based (the majority of Dreamweaver and Flash users). [3]

An informal survey of Universities from around the world confirms the impression that Multimedia is generally taught within Design / Arts Schools at undergraduate level, giving a bias towards non-technical, aesthetically driven approaches in those who have had exposure to the prevalent tools[1]. These students also tend to be the ones who go on to teach (as they tend to have stronger critical /theoretical / academic credentials) causing a self-reinforcing cycle[2].

While lecturing at the HRC, it was observed that technical students tended to be solution focused, frequently doing things to prove a technical capacity rather than to answer a design question. This approach led them away from more risky exploratory work and into safer technical demonstrations. Conversely, design students often concerned themselves more with the aesthetic aspects of a project than its feasibility, resulting in beautiful / interesting artifacts that were often less than fifty percent complete by project end.

As a result of the above observations, Nigel Power, then course leader at the HRC, asked this author to develop a syllabus that would bridge these two worldviews (technical vs. aesthetic).

The pedagogical goal of this course was to develop a curriculum, which, while accessible to students with both technical and non-technical backgrounds, would be full featured enough to permit the discussion of advanced programming concepts. This course was developed with a long-term view to improving the work of software design practitioners in the wider world.

The three primary areas of development methodology that we wished to demonstrate were/are:

1. The linguistic (as opposed to engineering) nature of programming
2. Object theory and object oriented development methods
3. Prototyping as a development approach

While many alternative tools exist[3], the concrete nature of Lego makes it an ideal vehicle for this exercise. As this paper shows, Lego exhibits many of the qualities that are required to illustrate the above aspects of programming methodology, while also having a capacity to present and solve design problems.

On top of this, MindStorms® also has a (largely) user-friendly programming interface that facilitates discussion of several software development concepts using a simple metaphor.

[1] Murdoch, where I currently teach, is unusual in having even the School of IT within the Division of Arts.

[2] A formal survey (by this author) is in progress. This survey looks at approximately five hundred Multimedia courses taught at undergraduate or postgraduate level, and the qualifications of the teaching staff on said degrees.

[3] These include programming environments such as StarLogo, Macromedia Director/Flash or, in the extreme case, X-Code. The difficulty with such tools is that they are already in the programming domain and, therefore, more intimidating to the non-initiated than Lego tends to be.

The assessment criteria of the course were:

1: The student's individual and group contribution to the task of building a Lego MindStorms® robot, which takes part in massed object avoidance with the other group's efforts.

2: The student's developed understanding of, and ability to demonstrate, multimedia development using a rigorous software design methodology.

2. Programming as a Linguistic Act

In order to understand the act of writing code as a linguistic process [8], the students are introduced to the relevant concepts in the following manner:

2.1.1 Languages (both natural and artificial) consist of a number of primitives and a number of rules governing the use of these primitives. [7]

In English, we have letters and words, with conventions around letter order within words and grammatical rules governing the arrangement of words into larger 'chunks' (phrases or sentences).

Similar linguistic rules to those used in spoken languages apply in most programming languages.

It is possible to demonstrate a trivial language using Lego bricks as tokens. The characteristics of this language (its rules, strengths, weaknesses, etc.) are then discussed.

2.1.2 The tokens or primitives of a language can generally be divided into a number of more or less arbitrary classes. [7]

In most spoken languages, one has verbs, nouns, pronouns, adjectives, etc.

In the programming domain, these generally consist of variables, keywords, events and functions or handlers.

In Lego, there are pieces of different types, distinguished by form or, rather, function. Therefore, you might end up with Connectors, Plates, Girders, Wheels, Cogs, etc.

2.1.3 In most programming languages (in fact, most languages), the pre-defined tokens are combined (iteratively) to produce statements which have a logical or agreed meaning within the language in question. [7]

In English, we combine letters into words, words into phrases, and phrases into sentences.

In a typical programming language, the primitives combine to produce logical statements, which combine to produce commands (a single line in most tools), which combine to produce functions.

In Lego, the individual Lego pieces play the role of pre-defined token. In a similarly linguistic manner, these can be combined in an iterative fashion to produce components that serve a logical, mechanical or design function. The MindStorms® programming environment also uses a modular approach, with predefined code blocks combining to provide the desired functionality.

Figure 1: Lego MindStorms® Programming Environment

2.1.4 These statements may be more or less well formed. According to the rules for the language in question, some combinations of classes are legal while others are not. [7]

One can make a grammatically correct statement that is difficult (or even impossible) to interpret without ambiguity in English. For example, "I saw her duck" – she bent over, or owns one?

So too with programming languages, if a statement is too ambiguous, or poorly formed, it will result in a syntax error.

In Lego, there are basic rules of engineering that determine whether a given solution can provide the required structural integrity or functional behaviour. Many of these rules are also present in other disciplines, so theoretically, at least, it should be possible to elucidate them. For example, a dick-atop-a-tom in dry stone walling has the same structure as a river in typography. This area provides a rich domain for upcoming work by this researcher and others. Equally, some combinations of components are simply not practical (for example, attaching a shaft to a plate requires a joiner of some kind – see classes, above).

2.1.5 Languages have varying tolerance for different dialects or modes of use (examples being variants of English vs. Strict STL C++).

English is an umbrella term for a huge number of dialects, although many are incomprehensible to an untrained ear. Thus, the language is highly tolerant of modification, but the modified result may or may not be portable.

The languages traditionally used in a multimedia environment are generally rather tolerant of different modes of use. Flash had very basic dialog driven scripting in version 2. The scripting improved in versions 3 and 4, but remained dialog driven. Version 5 added ECMA based ActionScript, with both modes supported until version 6 – a.k.a. MX. Macromedia removed the dialog driven method from the latest release of the software (v7 – MX 2004) and considerable (largely negative) comment has been generated in the user community as a result.

Similarly, in Lingo (the programming language of Macromedia Director), there are two dialects, verbose (which resembles English) and the newer Dot-Syntax, which uses largely the same primitives, but is structured more like Java or C++. In addition to the 'dialects' of Lingo, Macromedia have just added a whole new language to the application (JavaScript). While it is possible to mix development styles, it can become difficult to follow the logic if this happens.

In Lego, there are ways of using the available parts in different ways. For example the girders can be used like bricks, or like the cross members in a space frame. Certain design problems will also be much easier to solve if using the components in one or the other way, but, again, it can be difficult to work out what purpose a given part of the structure is serving if multiple modes are interspersed.

Figure 2a: Beams as Bricks

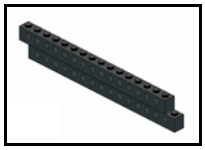
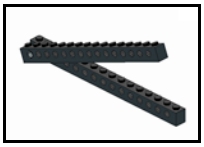


Figure 2b: Beams as Frame



2.1.6 Utterances or statements, of any given size, can be more or less elegant.

Throughout history, certain individuals have earned renown for their erudition.

In 'Andrew Rollings and Ernest Adams on Game Design', Rollings and Adams write that "Elegance is the sign of craftsmanship of the highest order". An elegant solution in programming is one that achieves the desired outcome in an efficient way. Efficient, in this case, means minimal code, minimal computational overhead and excellent human readability.

This is equally valid in the Lego environment, where elegance is one discriminator available when a number of solutions to a problem are possible. Elegance is often the main metric used when improving an existing design.

2.1.7 It is possible to create referential meta-statements in most languages.

Within a 'natural' language, single words or short phrases can take on a huge representational function. Examples are: 'Final Solution', 'Gay' or 'Disappeared'.

In both Lingo and ActionScript, as with most programming languages, you can create custom interfaces, via handlers or functions, which you can then refer to by name. This allows a very short interface statement (mDoSomeThing()) to serve a complex logical function.

Code example:

```
on mDoSomething  -- Declare Handler
     -- As many lines of code as you wish…
     -- …can go here…
     -- …including calls to other handlers
end mDoSomething -- Close Handler
```

Similarly, Lego allows complex interactions by virtue of mechanical interfaces, for example, the cog and shaft components. Here, a small, defined interface (the stub of a shaft) has a large effect (moving or modifying parts of the construction). [Diagram]

Within the MindStorms® programming environment, it is possible to create named procedures in much the same way as Lingo or ActionScript. These procedures can then be placed in the program as single items, rather than having to replicate the logic each time they are used.

2.1.8 Poetic or artistic license can give rise to unexpected outcomes.

In literature, poetic use of language takes advantage of the associations and hidden meanings of words. According to the Cambridge International Dictionary of English, Poetry is "…writing in which the words are chosen for their sound and the images and ideas they suggest, not just their obvious meaning".

Similarly, there are certain styles of programming in which these hidden results (known as 'side-effects') are used extensively. This is generally not considered a good development style, because, as with the literary version, such code is often difficult or impossible to read.

Code example:

```
on mDoSomething
     tReady = TRUE
     global_Total = global_Total +tReady
end mDoSomething
```

What is a number (global_Total) plus 'True' (tReady)? True normally equals 1 while false normally equals zero, but does this always hold (and is such code portable)?

Building a Lego artifact can be approached in a similarly obtuse manner. As with the literary domain, radical approaches are not always fruitful. Occasionally, however, something truly inspired can emerge from such an exploration. One student group produced a robot which transcribed a poem using pens held on with rubber bands. The initial discovery of the writing behaviour was serendipitous, but the final result was extraordinary.

3. Object Theory

The second aspect of development we were interested in describing was the idea of Objects, and of object oriented development methods [2][6].

Object oriented programming (OOP) is an approach to algorithmic and production methodology in software engineering. In fact, the methodology proposed in this paper has applications in any design discipline, not just software design.

OOP emphasises reuse, comprehensibility and design simplicity while attempting to achieve both development and runtime efficiency. According to Meyer, OOP has a number of attributes (five criteria and five rules), expanded upon below, which a system must possess or obey if it is to 'carry the badge'. [6]

All of these can be illustrated using the physical components from the MindStorms® kit, although a few concepts require the programming environment in order to demonstrate the practical advantage in using / following them.

The programming environment also allows a non-technical student to learn the concepts underlying solid development methodology with a very gentle learning curve (and was designed with this in mind). [4]

3.1 Criteria:

3.1.1 Decomposability:

Complex tasks are decomposed into sub-tasks.

3.1.2 Composability

Existing software elements can be recombined.

Due to the particulate nature of Lego, MindStorms® readily lends itself to the illustration of object based composition / decomposition. The individual pieces combine to produce subsystems, which interact / combine to produce the desired result.

3.1.3 Understandability

Code should be human readable.

Objects made from Lego components should, ideally, have a form which aids their application ("form follows function")[4]. Equally, MindStorms® programming blocks should be clearly named to indicate their function.

3.1.4 Continuity

Small specification changes require small code changes.

Changing the environment (room) in which the robots are expected to perform should not require a total rebuild. Perhaps the drive needs to be modified (to go from a tiled to a carpeted surface, for example) or the bumper may need revision (if the furniture has different heights/angles), but the bulk of the robot and code should remain unchanged.

[4] Sullivan, Louis H. *The tall office building artistically considered*. Lippincott's Magazine, March 1896. "Whether it be the sweeping eagle in his flight, or the open apple-blossom, the toiling work-horse, the blithe swan, the branching oak, the winding stream at its base, the drifting clouds, over all the coursing sun, ***form ever follows function***, and this is the law. Where function does not change form does not change. The granite rocks, the ever-brooding hills, remain for ages; the lightning lives, comes into shape, and dies in a twinkling." [emphasis mine].

3.1.5 Protection

Failure should be contained gracefully.

Becoming entangled in another robot, or objects / locations within the environment, reveals areas in which further design work is required. In the programming environment, the logic has to avoid thrashing (repeating a closed sequence of actions) when unexpected situations arise.

3.2 Rules:

3.2.1 Direct Mapping

Software should model the structure of the world it represents.

The structure of the robot components (for example, bumpers or drive) should map to their logical function. In code, this corresponds to code blocks having names and behaviours which map onto those functional components, and the environment.

3.2.2 Few Interfaces

Objects within that model should minimise the number of interactions they permit/perform.

The robot's components should be functionally and structurally independent. The modular nature of the programming environment constrains the possibility of interfacing between code 'blocks'. As such, this is implicitly demonstrated.

3.2.3 Small Interfaces

Object interfaces should transmit the smallest possible amount of data.

This and the following rule (Explicit Interfaces) are the most difficult aspects of OOP theory to convey using MindStorms®, for the simple reason that version 1.0 of the prgramming environment has only one variable, and uses pre-defined object interfaces. The physical Lego kit does offer a (poor) representation, in which changes to an interface (for example, the anchor for the bumper) can have a larger or smaller impact on the structure of the robot as a whole.

3.2.4 Explicit Interfaces

Interfaces should be declared and controlled in scope.

The physical kit allows anchor points and connections to be explicitly defined, and this illustrates this concept to a degree. Unfortunately, creating components with defined interfaces can make a Lego robot physically weaker than it would be if the whole robot were assembled as a single interlocked object. This is accepted as the price for the increase in ease of modification that results (physical robustness is not a quality metric in software). The payoff comes when a component needs to be modified - the rest of the structure remains as it was, generally only the specific component under review requires alteration.

3.2.5 Information Hiding

Data should be private, and accessed through defined, explicit, interfaces.

If the drive mechanism (physically or in code) shares a number of its parts with the bumper, then the two systems can interfere. As such, rather than having the bumper directly manipulate the drive mechanism, code blocks are created to abstract it (so "Bump Left" might trigger "Turn Right", rather than containing drive related commands/values itself). This also means that there is only

ever one place where the "Turn Right" behaviour needs to be defined or modified.

In fact, according to Meyer [6]: the five rules can be summarised as a sixth criteria:

3.1.6 Simplicity

The system, and each object within it, should be designed in the simplest way possible, with the minimum possible interface (exchange of data and public methods[5]).

4. Prototyping

The final aspect of development we explicitly try to demonstrate using MindStorms® is iterative development, or prototyping.

We found that this part of the course was particularly useful for the technical students. The designers were more often accustomed to using iterative development methods (design has been described as a conversation with materials). [5]

It is a frequent occurrence that the technically trained students would start looking at what appeared to be the most interesting part of the problem, though this may or may not be the most important or difficult part to complete.

Due to time and resource pressures, software developers are also often willing (or forced) to accept the first version of a function or application that works (or at least appears to in testing). This often leads to inefficient or buggy code, and explains the adage about never using x.0 releases for mission critical tasks.

In contrast, design training encourages the designer to explore multiple solutions and be consciously aware of the compromises and compensations they have made during the production process. Hence, students with a design background typically explore a greater number of possible approaches than those with a technical background. [5]

One of the advantages of object prototyping is that the developer finds and becomes familiar with the most difficult parts of the design or development problem early on in the process. This generally results in a better comprehension of the required effort to complete the tasks than a 'best guess' made at the beginning of the project and often leads to better and/or more robust solutions. [1]

Lego encourages this exploratory design approach, and helps to provide a concrete manifestation of each step in the process, highlighting where things might be improved.

To illustrate, the design of the bumper on a robot influences, and is influenced by, the design of the robot as a whole. Trying out different bumper designs may reveal flaws in the logic used to design the drive mechanism (too wide, narrow or high) or some other subsystem. At the same time, each iteration through the prototyping cycle leads to new insights into the design problem at hand. What may at first appear to be an insurmountable problem may in fact be easily solved, or what appeared to be simple may prove extremely difficult, or even impossible, to achieve.

Only by means of a loosely directed, rapid iteration, trial-and-error based prototyping process can a comparably thorough exploration of the problem domain be achieved so quickly. There is the option of Matrix Testing[6], but this requires very detailed advanced knowledge of the problem domain, which is not typically available in the student population.

5. Conclusion

The approach to teaching software design methodology outlined in this paper has been used successfully in a number of contexts. It has been presented to undergraduate and post-graduate students in classes with homogenous and heterogeneous backgrounds. In addition to several Multimedia/Hypermedia classes, it has been used with MSc Computer Science students and undergraduate Photography students.

While the metaphorical fit is not always perfect (some OOP concepts are difficult to demonstrate), we have found that the non-threatening nature of the tool, coupled with the clearly stated goals of the course, has lead to a very positive set of learning outcomes for those students who have participated in the programme. To illustrate, typically, the course would produce between one and three distinction candidates in any year. In the 2000-2001 academic year there were five.

While this course is only intended to provide the target group with a preliminary understanding of a specific software design methodology, student feedback[7] shows that they consider it an invaluable introduction, not only to the material covered, but also to the wider domain. This is particularly true for those students who go on to do hands-on programming modules subsequent to participating on this one.
As a result of encouraging technical students to explore the design process while designers look at new ways of

---

[5] Public methods are globally accessible functions or scripts.

[6] Matrix testing consists of an array of tests with each series in the test plan varying only one aspect of the system. Each test then has to be run with all combinations to confirm that there are no unforeseen interactions. The number of tests required for a complete matrix test rapidly approaches an unmanageable level if the system is complex (hence, the high cost of application testing in industry). There is an added problem, in that the individuals developing the test plan must be able to list all possible conditions under which the system is expected to operate, including those edge cases where failure is more likely. [Example at: http://jake.soapware.org/currentXmethodsResults]

[7] "...because of your lessons in programming and the philosophical structure behind it…I have started to gain the skills I need to implement my ideas. More importantly, the philosophy has had a broader impact in the way I negotiate the world."

- Rod Dickinson – HRC Student – private correspondence (with permission)

using their skills, both groups learn to excel in the software design domain through the mutual exchange of ideas and exposure to alternative approaches fostered by this dialogue.

The outstanding results consistently achieved by students following on from this course of study demonstrates that the deeper understanding of the domain fostered by the approach described is, as was hoped at the outset, leading to an improvement in the quality of work produced in the world by at least some of the software design professionals graduating from the programme.

6. Acknowledgements

I extend my gratitude to Nigel Power and Andrew Cameron (both formerly of the Hypermedia Research Centre at Westminster University) for giving me the opportunity to explore and expound the material outlined in this paper. My thanks also to the Australian Research Council and the Center for Social Change and Social Equity at Murdoch University for their support.

7. References

[1] Beck, K., Extreme Programming Explained. 2000, Upper Saddle River, NJ: Addison-Wesley. 190

[2] Cantwell-Smith, B., On the Origin of Objects. 1996, Cambridge, MA: MIT Press. 420.

[3] England, E. and A. Finney, Managing Multimedia, Project Management for Interactive Media. 2nd ed. 1999, Harlow, England: Addison-Wesley. 423.

[4] Knudsen J. Lego MindStorms: Lego and MIT, 2000 http://www.oreillynet.com/pub/a/network/2000/01/31/mindstorms/index1b.html

[5] Lawson, B., How Designers Think. 3rd ed. 2000, Oxford: Architectural Press. 318.

[6] Meyer, B., Object Oriented Software Construction. 2nd ed. 1997, Upper Saddle River, NJ: Prentice Hall PTR. 1254.

[7] Pinker, S., Words and Rules. 1999, London: Weidenfield & Nicolson. 348.

[8] Winograd, T. and F. Flores, Understanding Computers and Cognition. Vol. 1. 1997, Norwood, NJ: Addison-Wesley. 207.