Automating Software Design Pattern Transformation

Martin Wheatman, Kecheng Liu
Informatics Research Centre, PO Box 241, University of Reading, Whiteknights, Reading, Berks., RG6 6WB, UK
m.j.wheatman@reading.ac.uk, k.liu@henley.reading.ac.uk

***Abstract*-A Design Pattern has no one implementation; it is a linguistic device which allows the discussion of standard solutions with specific properties while avoiding implementation detail. Software designs using such patterns still require a skill-intensive transformation, taking into account the system technology and organisational requirements. One problem is that such manual transformations are less complex than configuring a program to generating the equivalent source code. Therefore programming largely remains an artisan activity. This paper applies linguistic analysis techniques to source code in an attempt to distil the design intention from the implementation detail. This enables the translation of patterns into source code: a mechanism is described and some simple examples are presented.**

I. Introduction

Design Patterns originated as a standard solution to common problems in town planning[1]. Software Design Patterns[5] also present similar conventional design components: the essence avoids implementation details. Further, the set of design patterns is open-ended: any agreed solution can be a pattern as long as it conveys the essence of the proposed solution.

While examples of implementations can be given, for example in [22][15], they by no means dictate *the* solution. Attempts have been made to link facets of source code to given design patterns, known as traceability, most notably [6], however this breaks the arbitrariness which is useful. This arbitrariness reveals the linguistic aspect in designing software - the implementations are a complex interplay between the intentions of the client, but also the application's technology and the organisational structure.

In attempting to automate source code production, this paper draws on the arbitrariness of a linguistic approach, making a distinction between the notion of translation (the mechanics of text processing – pattern to target token swapping, etc.), and transformation (the irreplaceable cognitive processes – the representation of intention, etc.). What this paper intends to show is how a subjective representation of a design pattern can be captured using the triadic sign model of [13], and how this can shift programming from an artisan activity to a manufacturing one. It starts by attempting to summarise some of the shortcomings of the software development process in the next section. In the third section, the linguistic models are described which attempt to describe textual representations, outside of the prescriptive dyadic software model. Following this, the mechanism is described and an example is presented.

II. Software Engineering

Much has been written on the process of software production including programming and design methodologies. This section starts with a look at how computers are employed to help manufacture source code.

A. Computer Aided Software Engineering

Best practice in the production of bespoke software is of a succession of methodologies (composed of an integrated suite of techniques) for eliciting requirements and organising these into various design models and components, with the final stages being the production of source code and the building and testing of binary artefacts. Each method of software production has an underlying philosophy (data oriented, process oriented, object oriented), and each seems to be an improvement on the last. As computers have grown in capacity, the support provided by computers themselves for each method has also grown, becoming an industry that reached a peak in the late 1980s - Computer Aided Software Engineering (CASE). Methods are prescriptive, and users tend to be selective with the components that are relevant to their industry, and their organisational requirement.

One attempt to break from the one size fits all approach was Meta CASE - allowing the CASE tool to be varied to match the organisational CASE requirements , e.g. [23]. The onus is on the organisation to develop their own method, and its support tools. Needless to say, this limits Meta-CASE to larger projects. Still, the emphasis is on the Aided - CASE does not do the programming itself - it still requires the appliance of skilled practitioners to perform the development and complete what is missing.

The next break from CASE is this unification of three of the main object-oriented method proponents: Booch, Rumbaugh and Jacobson. Their Unified Modelling Language (UML) [32] encompasses many of the techniques used in earlier methods, and since it also stresses the pick-and-mix approach, allows users to do so within the one method. The acronym CASE has since largely become defunct, in favour of Integrated Development Environments. Tools such as Eclipse [3] and Visual Studio [25] support the programmer in developing source code; and with support for UML being provided by the notion of plugins. This marks a shift from method support to programmer support; though methods are

978-1-4244-3760-3/09/$25.00 ©

still used, their silver bullet credentials have waned [8]. While CASE has often supported the generation of source code, this has often been limited to the generation of interfaces. Perhaps the evolution of CASE is to be found in [11] – providing the framework which can be populated with snippets of source code to provide a fully working program.

B. Frameworks

A brief mention is made here of frameworks, which include for the purposes of this paper, code generators such as the Java Server Pages servlet complier [34], and the Ruby-on-Rails notion of generating database connectivity software for given programming data structures – a notion which has been copied elsewhere, for example in CakePHP scaffolding. While these techniques have in common the notion of generating code specified by some other language, and as such can be seen as examples of meta-programming, they use various mechanisms to achieve their ends and are used in niche areas. Because of this, they resemble the automation of Design Patterns, as approached in this paper, but their application is limited to specific areas (particular languages and particular application areas). Indeed, Ruby-on-Rails is an implementation of the MVC design pattern [35].

C. Design Patterns

Design Patterns act at a more detailed level, describing commonly understood solutions to reoccurring problems: manifested as frameworks of classes used in software construction. They act at the level of designer-to-programmer interaction, giving a common understanding, forming part of the transfer of designs into source code: a designer can talk of a linked list, for example, and the programmer will understand the concept.

There is still the need to implement code – the programmer has to enact the intentions of the designer, has to choose how a pattern is to be implemented. For example, there are different implementations of the same design pattern, such as the Singleton[31] – which may be created at runtime or when needed (the lazy implementation). Also, the examples of design patterns given in Wikipedia, for example are all written in Java, but design patterns are not limited to a Java implementation. Further, implementation details are finer grained still: there is the need to be sensitive to the organisational context, for example the language of implementation, or the situation in which the linked list is to be used: is it private to a given process with sequential access, or is there it to be shared, in real-time, asynchronously between competing threads? It does mean, however, that the essentials are known: merely a linked list is required, rather than a doubly linked list or some form of tree structure.

Another attempt at establishing a link between the pattern and the implementation, pattern traceability, where the characteristics of a pattern can be seen in source code, are found in Micro Patterns [6]. This provides a simple measure of design: cataloguing the characteristics of certain interface designs – and thus being even closer to the implementation. A programmer is still required to transform design into code. Another attempt at cataloguing design patterns is given in [12].

D. Discussion

The conjecture is that although techniques are evolving to produce source code, somehow this evolution is slowing, which is reflected, if not explicitly stated, in comment such as [7]. The emphasis remains on improving the production of source code rather than moving on from source code itself: it remains the most flexible mechanism for representing instructions to a machine. Since the software industry is based upon the notion that it is technically easier to employ software engineers, that it is to generate that flexible mechanism, programming remains largely an artisan activity, rather than a manufacturing process.

The approach described in this paper does not move us on from source code, it will still be used in the manner that assembler is still used in operating systems and embedded applications. This paper concentrates on the mechanisation of design pattern implementation.

Design patterns cannot be directly implemented, they must be transformed. However, where we can identify translatable facets, we can remove this from the transformation process. What we attempt to remove is the intention of the designer from the implementation of the developer. The cognitive element, however, cannot be removed: Artificial Intelligence [20][16] has produced useful techniques for problem solving, but not the sought-after cognitive model envisaged in [21]. This is the approach in [33].

The importance of design patterns to this project is that they represent a portion of meaning shared between the designer and programmer. CASE imposes a structure, but often does not proceed beyond the level of interfaces. This project attempts to capitalise on the meaning of something to transcend an individual solution.

The philosophy underlying this is described in the next section, and a suitable mechanism is outlined in the following one.

III. Semiotics

Semiotics [4][2] is the study of signs: things which represent something else. It dates back to ancient times, such as the Craytalus dialogue [14], which ponders the meaning of names. However, two philosophers of the late 19th Century independently formalised systems of semiotics – Ferdinand de Saussure [17] defined Semiology in the early 20th Century, where a sign consists of a dyad of a physical Signifier (typically written or spoken) which refers, in an arbitrary manner, to Signified mental concepts. Individual acts of communication, what Saussure calls Parole, are part of all that can be said, Langue. Where two or more langues overlap, a language is formed. Semiology became known as

the European school of semiotics foundation for the 20th Century movement of Structuralism, and forms the basis for software systems today: an identifier refers to a value, a file name refers to a file, a function refers to it constituent components, a design refers to am implementation, and so on.

The second model was devised by the American philosopher Charles Sanders Peirce. His Semiotics (a word which has come to represent both schools) is a triadic model: "something which stands as something for somebody in some respect or capacity"[13, pp99]. Its three main components are: the Representamen - the physical sign vehicle, which can be anything that can be interpreted as a sign; an Object – that to which the sign refers; and most importantly the Interpretant – that which is generated by the interpreter in response to the sign vehicle. A sign only works through the generation of an interpretant, which is seen as an ostensible response – a reaction to the sign, rather than merely a private, mental image, as in Saussure's model. Finally, it is necessary to note that Peirce's model includes an interpretant being a representamen in its own triad. This deferrable meaning was also modelled by Hjelmslev [26], who worked in the Saussurian paradigm, as Denotation referring to some final meaning, and Connotation as referring to some other signifier/signified pair. This naming convention is also used in this paper.

The initial application of semiotics to information systems is [27], where Stamper models organisations as information systems. This is complemented by his model of information as sign [28]. Further approaches to semiotics emphasise the use of signs at the human computer interface [30].

This paper, however, is primarily concerned with source code as a textual representation of action. Saussure emphasised communities bound to languages, creating semantic gaps in-between: the analogy is that clients, designers and programmers use different languages, and the design process has evolved to pass the requirements of clients into software, through these communities. Through the interpretation of Charles Morris [29], and his triadic model of syntax, semantics and pragmatics, and his emphasis on the creation and destruction of signs, the Peircean approach involves the knowledge which remains in the system, known as norms. The approach in this paper is to utilise this information, to change the artisan transformation of programming into a plain translation of textual creation. As such, we are developing a general translation tool (stream editor) which is policy free - the emphasis is not on the generation of source code, but translating portions of text.

Finally, Saussure was only interested in writing and speech. Peircean Semiotics is concerned with all manner of signs noting Ground as the medium within which the sign system works, and this is what we look at next.

IV. The Mechanism

The language independent mechanism used in this project is believed to be novel, and has several advantages, not least enabling the translation of any programming language: reducing the onus on the generator manufacturer to support particular languages.

A. Sign Representation: Ground

The medium used to represent signs in source code is borrowed from the markup tags of the SGML family. This is not XML: the choice stems from the ability of the markup family syntax to intersperse one syntax within another. This mixing of syntax is similar in principle to the mixing of comments within source code defining how that code should be interpreted by programmers reading the code; however, to use such a mechanism of comments would be language dependent.

Further, there is the raw ability of angled brackets to directionally quote string values, and intersperse tags and their attributes, in a way that the double-quote has difficulty: given a page of double quoted values, it is difficult to see what is inside and what is outside quotes. Thus, in the string:

"b "a" b "a" b "a" b "a" b "

each a is outside the double quotes.

B. Sign Decoding

The process of translating a sign into an object is that of decoding. Basic examples can be seen in the search and replace facilities of word processors; the token replacement of stream editors such as sed; or cut and paste functionality in tools such as Maven. In terms of source code production, the C pre-processor, cpp, effects this type of processing, and is a powerful tool, providing: token replacement and parametrised token replacement; file inclusion; and conditional compilation. In some respects this project still has to be developed to reach this functionality (see Further Work), but the use of such preprocessing techniques is not encouraged, particularly in Java where there is no preprocessor (possibly the JSP compiler?), and ultimately it is a language-dependent tool.

The mechanisms so far developed for sign decoding are summarised, thus:

TABLE I
Denotational Translation – Basic Substitutions

Representamen	Interpretant	Object	Description
<a/>	<a>b	b	Forward substitution
<a>b	<a/>	b	Reverse substitution
	<a>	b	Pattern/target substitution

TABLE 2
Connotational Translation - Implication

Repr.	Interpretant	Object	Description
<a b/>	c<a/>	c	a implies b, b substitutes as c. Attributes can be supplied in a value thus b='d="e"'

These first two constructs outlined involve simple decoding and token replacement which could also be achieved by cpp or sed. The following transformation mechanism, however, to the authors' knowledge, cannot be produced by simple token substitution.

TABLE 3
CONNOTATIONAL TRANSLATION - ORDERING

Repr.	Interpretant	Object	Description
<c>	1	123	b substitutes as 1;
<a/>			a implies b;
</c>	<c>		c substitutes in a;
	<placeholder name="b">		place holder to
	2<a>3		order b before a.
	</c>		

Other mechanisms, perhaps including those provided by cpp, will be required as the project develops, and at least one is outlined below, however, these are being implemented on a practical basis.

C. Hello World Example

The following examples are all working and have been taken from the test suite of the primary author's PhD.

The first example given here is the Hello World program [10]. It is the standard example program, given in many texts on programming languages, showing that a new installation of a compiler is working. It demonstrates the three translational mechanisms described above.

Firstly, because it can be recognised by many programmers, and because it is represented in a particular way in source code, it is argued here that it is a traceable pattern in itself (but not necessarily a design pattern.) As such, it can be represented in its entirety, as a sign vehicle thus:

```
<helloworld/>
```

Given an interpretant of

```
<helloworld>
#include \<stdio.h>

int main()
{
       printf( "hello, world\n" );
}
</helloworld>
```

it demonstrates the ability of the sign <helloworld/> to be translated into the complete hello world C program, in the manner of a standard stream editor.

However trivial the Hello World program is though, it fully exhibits the interconnectedness of source code text: a more insightful example of a sign vehicle would be:

```
<prog><print param='"hello, world\n"'/></prog>
```

Given an interpretant of:

```
<include>#include <file/></include>
<print
include='file="<stdio.h>"'>printf( <param/> );
</print>

<prog><placeholder name="include"/>

int main()
{
    <content/>
}</prog>
```

would also produce the Hello World source code. The main sign vehicle being the <print/> vehicle within the <prog/>. This is used as the content in the interpretant's <prog/> vehicle. The print sign is further decoded. The existence of the include tag attribute necessitates the decoding of the interpretant <include/> sign with the given parameters: one sign implying the existence of another.

Furthermore, the use of the <placeholder/> sign vehicle in the interpretant's <prog/> sign indicates the ordering of signs, as part of the transformation from sign vehicle to artefact.

The second example does not analyse the notion of C strings and parameters further - it assumes the programmer understands these, and may be an irreducible minimum.

A further example of interpretant is the following:

```
<print>System.out.print(<param/>);
</print>
<prog>class helloworld {
  public static void main(String argv[])
  {
   <content/>
  }
}</prog>
```

This example shows the importance of language independence, whereby a Java hello world program can be generated from the same sign – the implementation is down to the interpretation of the designer's signs.

D. Current Work- Further Examples

While the hello world is a complete working example which demonstrates the current functionality of the decode tool, specific design pattern examples are briefly given here for illustration. Firstly, there is a factory method, an instance of which is defined as a sign:

```
    <factory type="Pizza">
     <product name="HamAndMushroom"/>
     <product name="Deluxe"/>
     <product name="Hawaiian"/>
    </factory>
```

A Java interpretant for this could be:

```
  <import>import "<product/><type/>";
   </import>
   <placeholder name="import"/>
   <productTypeList start="{" sep="," end="}">
      <name/>
   </productTypeList>
   <productCase>
               case <name/>:
                 return new <name/><type>();
   </productCase>

   <product productTypeList productCase/>

  <factory>
   class <type/>Factory {
    public enum <type/>Type <productTypeList/>
    public static <type/> create<type/>(<type/
>Type t) {
      switch (t) {
           <content/>
      }
      throw new IllegalArgumentException("The
<type lower/> type " + t
     + " is not recognized.");
    }
   }
  </factory>
```

The notion of a list has not, as of writing, been implemented; though now identified, it is currently under construction.

E. Future Work

Currently, this project is only able to generate code (deductive decoding), and initially, future work includes the expansion of this, as highlighted above: in a language independent manner.

The ability to encode signs falls into two categories – firstly, there is the creation of signs from source code given the interpretant (deductive encoding). The ability to achieve this may depend on whether the source code has been generated by the decode tool (some indication may need to be given as to how the decode has proceeded), but the goal would be able to work on existing code. However, this is seen as being of a similar level of complexity to decoding.

The second, more difficult encoding, is that of creating interpretants (inductive encoding), which may yet prove to be impractical. It is envisaged that it may be possible to determine candidate interpretant patterns, which will require some cognitive input on intention.

Lastly, there is also the need to represent this in some graphical form – rather than as raw markup text, but this has a much lower priority.

V. Conclusion

Design patterns are important to this project as they are arbitrary portions of code that mean something to the designer and programmer: not limited to the structures the programming language provides, e.g. interfaces, classes. Further, they should not be limited to the macro patterns described in the current sources: smaller patterns can be identified, which would increase the coverage of code. There should be no reason why the same processing cannot be performed on textual representations of requirements and design, save from the use and subsequent impenetrability of Microsoft Word formats.

The generation of source code is trivial. The complexity of generated source code, though, is bound to the domain of structures that the code generator produces. The complexity in this code generator is intended to be open-ended as the techniques generated are the responsibility of the organisation generating the code, not the code generator supplier.

The approach presented in this paper takes the objective model of subjectivity described by Charles Sanders Peirce, and applies it to the process of design to source code transformation. In that an interpretant is an ostensible reaction to a sign, the source code in the meaning plans are certainly textual representations of action.

Inasmuch as this approach resembles meta-programming, since it generates source code, other similar approaches have been taken by others, and a list can be found on [24]. A review of these approaches is in order, but out of scope of this paper. While the presented approach resembles Intentional Programming, the approach here is orientated towards semiotics, and a policy free – language independent (unique feature!).

The tool described in this paper does more than the C preprocessor and stream editors such as sed (to the author's knowledge). The C-preprocessor is a simple but powerful tool, and this project recognises the power of the ability to textually process code. It remains unable to pre-process code in a structured manner, such as the substitution of the `#include` and the `printf()` together-but-apart in source code.

This project is by no means a complete solution to the mechanisation of source code creation, but the approach described this paper distils intent from implementation, freeing the implementation to be mechanised. Designs contain intentions of the requesting client, the resultant software should also contain these intentions. However, this intent need not be passed through to the programmer, if the

programmer can supply the facilities to support that intent up to the designer.

Acknowledgment

M.J.W. would like to thank Ronald Stamper for the inspiration for this work.

References

[1] Alexander, C., Ishikawa, S., Silerstein, M. (1977) *A Pattern Language: Towns, Buildings, Construction*, OUP
[2] Chandler, D. (2001) *Semiotics: The Basics*, Routledge
[3] see http://www.eclipse.org/
[4] Eco, U. (1977) *A Theory of Semiotics,* Indiana University University Press, Bloomington.
[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994) *Design Patterns - Microarchitectures for Reusable Object-Oriented Software*, Addison Wesley, Reading, MA
[6] Gil, Y., Maman, I. (2005) Micro patterns in Java code, Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications, *ACM SIGPLAN Notices*, Vol. 40, Issue 10 (Oct 2005) pp97-116.
[7] Glass, R. L. (2005) "The First Business Application: A Significant Milestone in Software History" *CACM.* Vol. 48 No 3, March 2005 25-26.
[8] Glass, R. L. (2005) "Silver Bullet Milestones in Software History", *CACM.* Vol. 48 No 8, August, pp15-18
[10] Kernighan, B., and Ritchie, D. M. (1988). *The C Programming Language, Second Edition*. pp6 Prentice Hall, Eaglewood Cliffs New Jersey.
[11] Selic, B. "The Pragmatics of Model-Driven Development", *IEEE Software*, September/October, 2003
[12] Noble, J. Biddle, R., Tempero, E. (2001) "Metaphor and Metonymy in Object-Oriented Design Patterns", 25th Australasian Computer Science Conference. *Conferences in Research and Practice in IT* Vol. 4, Melbourne, AU
[13] Peirce, C. S. (1955) Semiotic as Logic. *The Philosophical Writings of Peirce*, Ed. J. Buchler, Dover Publications, pp 98119.
[14] Plato, (1997) *Complete Works*, Ed. Hutchinson, D.S., Hackett Publishing, Indiana
[15] Pree, W. (1994) *Design Patterns for Object-Oriented Software Development*. Addison Wesley, Wokingham, UK, Reading MA, Menolo Park, CA.
[16] Rich, E. (1983) *Artificial Intelligence*, McGraw-Hill
[17] Saussure, F. de. (1983) *Course in General Linguistics. Ed. 3*. Eds. Bally, C., Sechehaye, A., Riedlinger, A, 1915, transl. Harris, R. Duckworth London
[19] Stamper, R. K. (1985) Towards a Theory of Information: Information: Mystical Fluid or a Subject for Scientific Discourse, *The Computer Journal*.28: 195-199
[20] Winston, P.H. (1984) *Artificial Intelligence: 2nd Ed.*, Addison-Wesley
[21] Turing, A.M., Computing Machinery and Intelligence, *Mind*, 59, pp433-460, 1950
[22] see http://en.wikipedia.org/wiki/Design_pattern_(computer_science)
[23] see http://en.wikipedia.org/wiki/MetaCASE_tool
[24] see http://en.wikipedia.org/wiki/Meta_programming
[25] see http://en.wikipedia.org/wiki/Visual_studio
[26] Hjelmslev, L. (1970) *Prolegomena to a Theory of Language*, University of Wisconsin Press.
[27] Stamper, R. K., *Information in Business and Administrative Systems*, BT Batsford, London, 1973
[28] Stamper, R. K. (1985) Towards a Theory of Information: Information: Mystical Fluid or a Subject for Scientific Discourse, *The Computer Journal*, 28, 195-199
[29] Morris, C. W. (1938) *Foundations of the Theory of Signs*, Chicago University Press, Chicago
[30] Andersen, P. B. (1997) *A Theory of Computer Semiotics*, Cambridge Series on Human-Computer Interaction, CUP, Cambridge, UK
[31] http://en.wikipedia.org/wiki/Singleton_pattern
[32] Fowler M. (2003) *UML Distilled: A Brief Guide to the Standard Object Modeling Language (Object Technology Series) 3rd Ed.* Addison Wesley.
[33] Winograd, T., Flores, F. (1986) *Understanding Computers and Cognition*. Addison Wesley
[34] Hunter, J., Crawford, W. (2001) *Java Servlet Programming, 2nd Ed.* O'Reilly.
[35] see http://en.wikipedia.org/wiki/Model–view–controller