Writing Code for Other People
Cognitive Psychology and the Fundamentals of Good Software Design Principles

Thomas Mullen
tom@tom-mullen.com

Abstract

This paper demonstrates how the cognitive model of the mind can explain the core fundamentals behind widely accepted design principles. The conclusion is that software design is largely a task of chunking analogies and presents a theory that is detailed enough to be accessible to even the most inexperienced programmer. The corollary of which is a pedagogical approach to understanding design principles rather than the necessity of years of software development experience.

Categories and Subject Descriptors D.2.2 [**Design Tools and Techniques**]: Object-oriented design methods. J.4 [**Social and Behavioral Sciences**]: Psychology– abstract data types, polymorphism, control structures.

General Terms Design, Human Factors, Theory.

Keywords chunking analogies; 4 minus analogies

1. Introduction

Making code easier to understand is the primary driver behind the phase of software design that does not change how the code behaves or performs. Design principles and software languages have evolved over generations to influence code authors to produce code structures that are easy to understand. At the same time psychologists have been unlocking the secrets of how the mind learns and understands. Later sections will illustrate the many similarities between the cognitive model and widely accepted design principles. This will serve as supporting evidence for the claim that code produced using such principles & languages is a textual representation of the memory structures within the brain.

The mapping between cognitive psychology and software design leads us to the discovery of the importance of analogies. If Structured Design is decomposition of the problem using the discipline of chunking, then Object-oriented Design is decomposition using the discipline of identifying analogies.

OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA.

1.1 The Need to Learn

A major portion of the time spent coding and designing is taken up in learning and understanding the application code. This is driven by:

1. The majority of the development cost is spent maintaining and extending code.
   a) Software applications typically last many multiples of the time it takes to develop the initial version.
   b) For successful applications it is common for team size to grow right up to the point that it becomes legacy.
2. Access to the original designer/author is uncommon.
   a) The lifetime of a software application will span many cycles of staff turnover.
   b) Global development of applications is becoming increasingly common. This includes regions with little or no window of common work hours.

Consequently, most software tasks are to extend/mend existing software where any understanding must be gleaned directly from the code.

Further (ignoring incapable or saboteur programmers), most bugs and deficiencies can be put down to failures in either understanding the requirements or understanding the existing code. A programmer who has complete familiarity with both the code and the requirement is most likely to establish the full extent of changes and the full extent of the impact of any changes on the existing behavior.

There is a great cost involved in learning/understanding the code and potentially further costs for fixing the issues that arise due to its misunderstanding. Where we have a

choice, investing in writing code that is easy to understand will produce efficiencies and cost savings throughout the life of the application.

1.2 Current Texts

Many texts on design principles implicitly require the reader to possess a certain level of experience in order to apply the principles appropriately.

When experts pass on their design skills, it is easy to fall in the trap of using an explanation that appears to them to be a natural consequence but, in fact, leans on the very same experience that guides them when executing that skill. Telling a child who is unfamiliar with a keyboard layout that the semicolon key is next to the "L" key is of limited use as a guide because finding the "L" key requires the same searching strategy as finding the semicolon key. When the touch typist hits the "L" key it is done automatically, with very little conscious notion of how the finger arrived at the key. The process seems a perfectly natural one. The guide of being next to the "L" key is useful only to the expert. The irony is that it is the novice that has most need for a guide. In Martin Fowler's excellent Refactoring book [9], a guide on when to employ the "Extract Method" rule is "if the method is too long or the code needs a comment to understand". The ability to recognize that the method is too long (or where it needs comments) is a product of the same experience as the ability to employ the rule appropriately. To maximize the value to the novice, the guide should rely solely on the limited experience they possess (e.g. the semicolon is on the middle row of letters to the right).

2. Cognitive Psychology

The aspects of cognitive psychology that are pertinent to our cause are discussed in the following sections. The main elements are as follows:

1. Chunks - Gobet et al [12] define a chunk as "a collection of memory elements having strong associations with one another but weak associations with elements within other chunks".

2. Short-term memory (STM) - also known as working memory, STM holds the items that have current focus (e.g. when solving problems) and is also responsible for the formation of chunks to be committed to long-term memory (via rehearsal). STM is limited by capacity [17] and time (<10secs without rehearsal).

3. Long-term memory (LTM) – a seemingly infinite resource. LTM is where we store our memories and experience ready for recall into the STM when solving problems. The only known restriction to LTM is the time that it takes to record (2s -8s).

4. Expert knowledge - experts and novices differ in the way they approach and solve problems. More crucially for design, a code structure that improves clarity for the novice may have a detrimental effect for the expert [22].

5. Analogical reasoning - analogies are ubiquitous in human intelligence [14]. Identifying and choosing analogies is driven by similarities, structure and purpose [15].

2.1 Chunking and Memory

Mathematical analysis of memory networks has shown that searching is optimized when the nodes are chunked to four or five elements [8] (there is an argument that the STM capacity limit is an evolutionary choice to force production of optimal long term memory networks [16]). Overloading STM will either necessitate dynamic chunking by the mind or will cause a sense of confusion. Chunking our code into groups of four or fewer elements (either visually or using language structures) means that subsequent readers are less likely to suffer cognitive overload of STM and the confusion it can bring.

When we try to understand a concept or find a solution to delivering a requirement, the mind naturally chunks related elements together. This is not necessarily immediate and may be the result of trial and error. The stronger the association between two elements (e.g. two methods that use the same fields) the more likely they will be chunked together. This chunking characteristic has parallels in many design principles. The Data Object is a collection of fields that are related together. Data normalization is a process driven by the desire for chunking using classes and/or database tables to partition the chunks. Although it is not their sole purpose, packages can be used to chunk classes, classes can be used to chunk methods/fields and methods can be used to chunk statements. In advising that a class have no more than 2 or 3 collaborators, Beck and Cunningham's CRC model is chunking the view of relationships to no more than four elements. Many of Fowler's refactoring rules are strategies to chunk elements (some are identified in section 4.2).

The laws of Prägnanz, from the Gestalt branch of psychology, identify how we recognize groups of elements. Many code authors employ these laws to indicate the element chunks so as to pass on the knowledge of the associations (thus saving the reader from the same, potentially costly, process of identifying them). For example:

- Chunking using the law of proximity. In the example below, statements are visually grouped together using blank lines. Lines that are close together will be chunked together by the reader.

```
printHeader();
printMsg();

processNew();
```

```
update();

printFoot();
```

- Chunking using the law of similarity. Indented lines are associated together due to the similarity of their shape. In the example below the statements executed as part of the loop will be chunked together by the reader.

```
sum=0;
sumOfSquares=0;
for(int I=0; I<num; I++)
{
    sum += x[I];
    sumOfSquares += x[I] * x[I];
}
```

Chunking code elements can be likened to grouping magnets that are sometimes attached by springs. There is a repelling [magnetic] force between all elements to prevent them being grouped together when there is no association. The associations [springs] are attractive forces (stronger associations are represented by stronger springs). The strong associations will bring some elements together and their combined magnetism will repel elements that have weaker (or no) associations. For example, if a class evolves so that one half of the methods use one portion of the fields and the other half the remainder then the stronger associations within each of the groups will lead to the pairs separating and provide an argument for splitting up the class.

Many texts already identify that the decomposition associated with structured design is a reflection of the chunking process which the mind employs to understand the problem. The guide that we should maximise cohesion and reduce coupling is the optimization function that produces chunks as defined by psychologists ("a collection of memory elements having strong associations with one another but weak associations with elements within other chunks").

2.2 LTM structure and relearning

The storage of items in LTM has been successfully modeled as "discrimination" nets [11]. Discrimination nets have been used, among other things, to model decision making, concept formation and recognition processes. The theory proposed that elements of memory are built up into a connected network. For each element to be added to the net, the place where it is to be incorporated is firstly identified. The net is then either extended or modified to allow the new data. In addition to the parent/child links of the network, each node has an associated "image" (letter, word, sound, visual image, feeling, etc). Anyone familiar with mind maps will immediately recognize this structure.

Building up the networks in LTM is achieved by "rehearsal" (after they have been loaded into STM). The sole cost of LTM is purely in the amount of time it takes to successfully rehearse elements (typically of the order of seconds). However, new elements may restructure the net so as to break the links to existing elements. These existing elements need to be re-learned to allow the building of a network that accommodates all the learning. Naturally, any relearning requires additional time costs (and sometimes frustration on the part of the student who is annoyed at themselves for seemingly going backwards).

Meyer's open-closed principle is the echo of the mind's process to minimize re-learning costs. This makes evolutionary sense. For example, if I see someone being violently ill after eating a black and yellow lizard I stand a better chance of survival (and passing on my genes) if I remember to steer clear of such reptiles. If I subsequently see someone feasting heartily on a black and yellow snake with no after effects, I will want to remember that snakes are good to eat without modifying the knowledge that lizards are dangerous. I will want my memory to be open for extension but closed for modification.

Our minds have evolved so that they structure LTM to optimize searching and minimize re-learning costs. If we structure our code to mimic LTM then it will be more easily and quickly absorbed (and understood) by subsequent readers. Design principles are symptoms of how the mind works rather than rules based on mathematical algorithms.

2.3 Simplified Cognitive Model

This section will detail a (very) simplified model of the cognitive elements (see Figure 1). Without wishing to become involved in the vigorous debate [5] on the capacity of STM the assumption will be that the limit is four chunks.

The gateway to an expanse of information the size of a planet (LTM) is a four window portal (STM). Adding to, or retrieving from, LTM can only be performed through the four windows. In something akin to Google Earth, the windows can contain big items (countries, states, cities) or zoom in to fine grain items such as words on a book. However, only one item (chunk) can be pulled in to each window at any time and, unless rehearsed, they will float back down to the web of LTM and soon (typically <10sec) disappear from our conscious.

The topics we desire to learn are interconnected elements like balls of spaghetti. In traditional topics, such as physics, teachers unravel and reshape the complex connections and feed it to students so that it fits through STM and has a good chance of reshaping into something useful on the other side (LTM). For the software application, design principles influence the programmer to create a spaghetti ball that is already reshaped and unraveled. Indeed the ideal situation would be that the application could be simply poured through the portal, where the only limitation was the flow rate (the time taken to commit to memory). The less complex the translation between the software code and the structure of the LTM

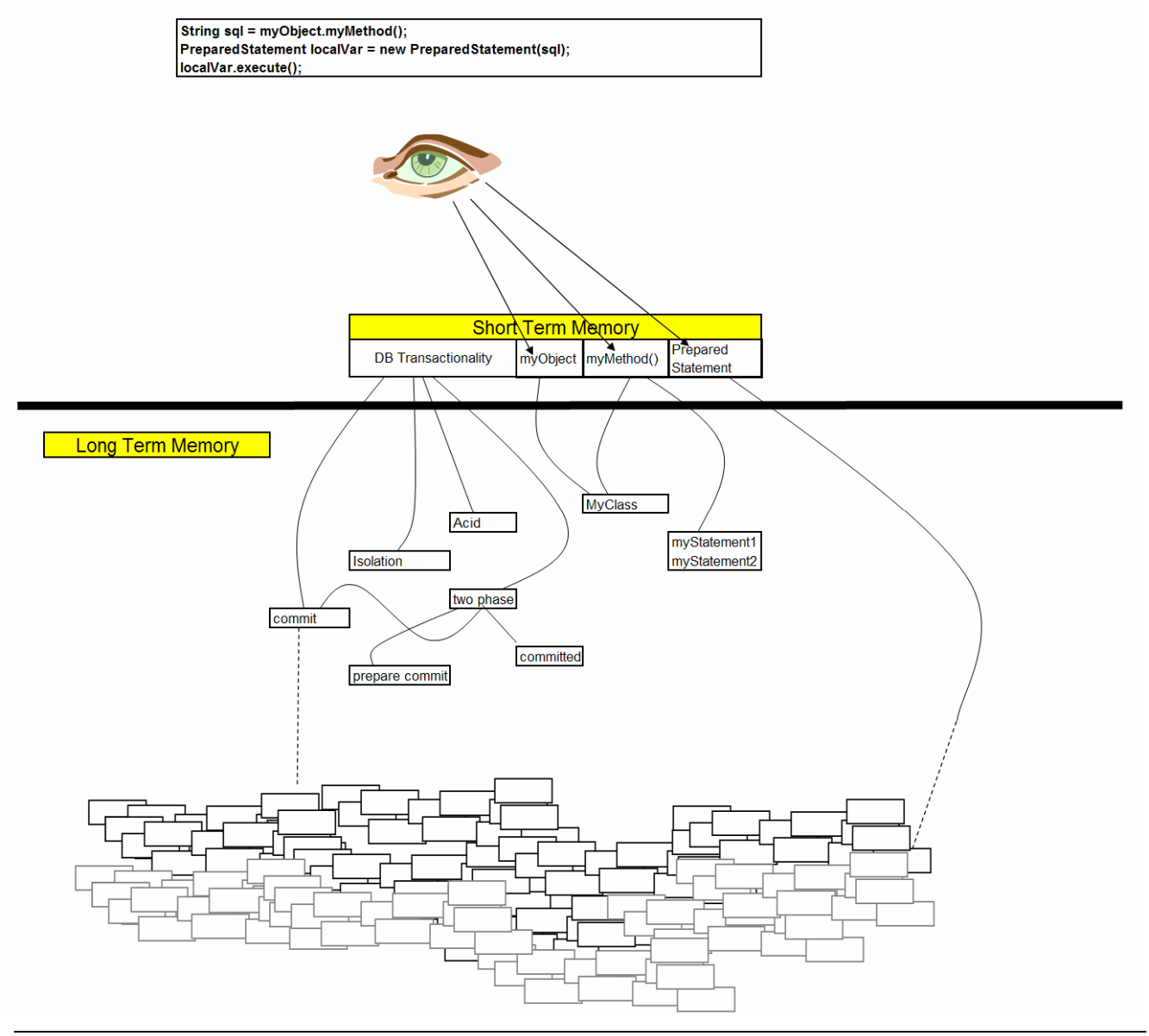

Figure 1. Simple Cognitive Model.

network, the less likelihood of mistakes by a new reader in formulating (i.e. understanding and learning) and therefore the less likelihood in needing to restructure their memory network (and the associated possibility of re-learning being necessary).

2.4 Evolution

Fred Brooks "No Silver Bullet" paper [2] talks about the essential complexity of the problem and the accidental complexity that we may bring to the solution by our choice of language (e.g. assembler) or design. Brooks surmised in 1986 that the current high-level languages have evolved to their limit. If we also surmise that the evolution of languages and design principles has been driven by the desire to make code easier to understand (as that is where the biggest influence of cost is), then by Darwinian argument:

CONJECTURE Current software languages and design principles guide a programmer to produce code that is a direct textual representation of the memory network of the solution within the brain (subject to the constraints of short term memory).

2.5 Cognitive Load

In studies of cognitive load for the effectiveness of training strategies [3], two principles of note are "Redundant Information" and the "Split Attention Effect":

If information is added that simply restates existing points and adds no extra insight, the concept is more difficult to comprehend/learn. The redundant information is not necessarily benign; it may take up scarce STM resources, which leaves less capacity to understand the intended concepts. For example, the comments associated with a simple getter method will usually just restate the method signature (there is no further insight to add). These comments can *add* to the complexity of the code.

In the split attention effect, if text that supports a picture is presented separately from the picture it is more difficult to comprehend/learn than if the text were displayed meaningfully upon the picture itself. In this instance additional items in STM are required to keep tracks of the links between the text and the picture. This leaves less capacity to comprehend/learn the concepts. Adding layers of indirection is a tool often used by the software programmer (for example chunking code lines into a separate method as in the extract method in Fowler's "Refactoring"). In doing so, however, we are increasing the cognitive load on the code reader, as they are additionally burdened by the newly introduced links.

Traversing a layer of indirection in the code may have both time and capacity penalties for STM. For example switching to another class to examine the workings of a called method may take a few seconds (seconds count with STM). Each level that is traversed may need to be understood (including peripheral items), taking up STM capacity and pushing the original contents out where they can no longer be rehearsed. We are remarkably adept at chunking all this information to keep a few levels still in STM. However, there is a limit, for which there will be no warning. Just a realization that we no longer remember how/why we got to this part of the code.

Each indirection appears not to make the code overly complex, as, when looked at in isolation, the additional burden is no more than other pre-existing indirections. Unless we are frugal, it is all too easy to breach the limit that results in confusion when traversing the path. IDEs can help to reduce the cognitive overhead (e.g. quick views on methods etc) as can manual memory paging (writing down). Whilst there are benefits to employing indirections (e.g. for chunking) the cognitive model also identifies a cost.

2.6 Code for Experts and Novices

The more expert we are, the more we scan the code rather than read it. The patterns in the visual area are processed and matched with templates in memory that have built up over the months and years of our experience. These templates allow the expert to immediately "see" the structure (as well as anomalies) as if the processing was done as part of the subconscious [7]. For experts it is therefore more beneficial to have as much code as possible in the field of vision (the split attention effect is also reduced). However if too much code is placed in the field of vision of the novice then cognitive overloading is possible. Novices benefit from the code being structured to direct their attention to a few items at a time.

Expertise of part or all of a software application is not restricted to programmers with vast experience. Certainly, a general software expert will find it easier than a programming novice to pick up a new software application. However, each of us (novice or expert) immediately becomes an expert in any code that we write by the necessity of having to fully understand the problem and solution in order to get it to work. Also, the software application as a topic is small and well contained when compared against conventional topics such as Physics etc. Consequently it may take only a matter of months or a year before even an inexperienced programmer becomes expert in part or most of the application.

As an example, novices will typically prefer a more aggressive normalization than experts as this provides them with the chunking of the data and thus reduces cognitive overloading. The expert, however, has enough knowledge templates in memory to "see" the chunking and will perceive such normalization as a layer of indirection that serves no purpose and indeed gets in the way of understanding.

There will always be the contradictory aims of keeping as much in the visual field for the expert and formally chunking to reduce cognitive loading for the novice. There must be tolerance on both sides. It won't take long for the novice to become an expert in the application when they will benefit from an expert layout. Conversely most teams have significantly more novices than experts so the "greater good" may mean that the expert should suffer a heavier indirection overhead attributed to chunking.

When coding we may start with a base that is clear and simple, but as we continue to implement the full requirements we will append code on to that base. Being an expert in the code that we write means that there is a natural creep to make more code present in the visual field. Code reviews (in addition to other benefits) are useful for alerting us to complexities that as authors we become oblivious to.

2.7 Analogical Reasoning

To many psychologists, analogies lie at the heart of human intelligence. Some reveal new insight (e.g. Niels Bohr's model of the atom as a small solar system) and others allow us to transpose existing skills to new tasks (e.g. being able to pilot a motor boat when we have only ever been trained in driving a car). In his presidential lecture [14], Hofstadter sees analogy at the core of cognition and illustrates their ubiquity. Analogies both facilitate the leaps in humankind's understanding and are at the very heart of our everyday cognitive processes. They are the building blocks of how we view concepts.

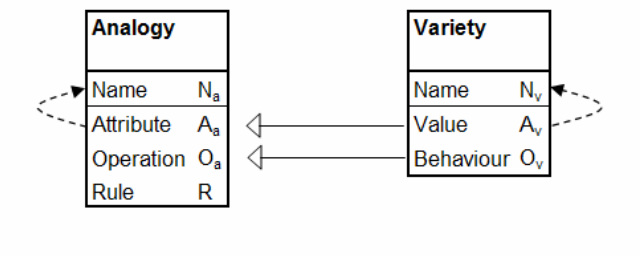

Figure 2. Analogy structure.

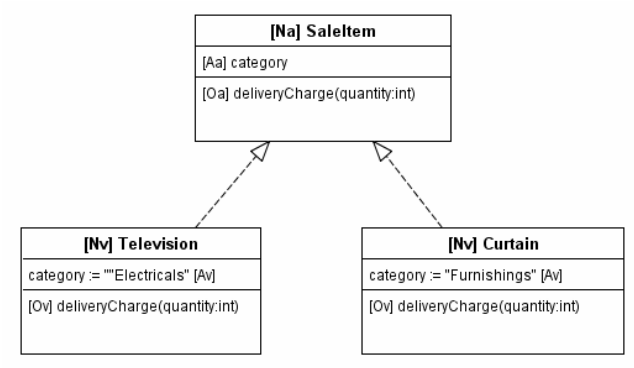

Figure 3. Example Class Hierarchy to illustrate the elements of an analogy.

Holyoak & Thagard [15] identify three broad constraints in choosing analogies - Similarity, Structure and Purpose. For example, a games programmer may model a tiger's behavior dependent upon its state, e.g. hungry, injured, cornered etc. The tiger analogy in this case has the different states as the varieties and the structure elements such as movement and likelihood of attack. However, when the World Wild Fund for Nature (WWF) looks into the threat of extinction to the tiger the structure elements are environment, breeding cycles, current population etc. In the WWF analogy the tiger is a variety and is similar to other endangered species such as the panda. It is purpose that determines the choice of analogy for the games programmer and the WWF. In software, the same class or code element may also present different analogies to separate parts of the code. For example, a data access object will present a read/write behavior to the elements of code that need access to the data but will present a "setDataSource" analogy to the elements of code responsible for initialization. The choice of analogy by each element of code is driven by the purpose that element wishes to access the DAO. Presenting one analogy to the business functionality whilst hiding other aspects of the DAO may mean that cognitive load is reduced when understanding the business logic.

In 1983 Gentner [10] proposed an algorithm for how the structure of the analogy influences choice. Analogies with similar operations are preferred to those that are simple similarities in attributes.

Gentner categorizes analogies by the types of mapping between the varieties. These can be mappings between attributes or relations (a relation can exist between attributes and/or other relations). The categories are reproduced here (two of the names have been changed to avoid confusion with software terminology):

	No. Of attributes Mapped	No. Of relations mapped	Example
Literal Similarity	Many	Many	The K5 solar system is like our solar system
Vanilla Analogy	Few	Many	The atom is like our solar system
Rule Analogy [1]	Few	Many	The atom is a central force system
Anomaly	Few	Few	Coffee is like the solar system

[1] Rule Analogy differs from vanilla analogy and the other comparisons in having few object attributes in the varieties.

For example, the instances of a class are literal similarities (the class description itself serves as the mapping) as are the rows of a database table (the column names serve as the mapping). Interfaces and abstract classes can be used to detail the mapping for vanilla analogies. The need for late binding and/or loose typing is usually associated with a rule analogy (for example, the Set class does not know what type of objects are to be placed in the set).

2.8 Analogy Patterns in Software

The psychologists view of analogies [10] (i.e. having attributes and operations) is similar to to the class/interface structure evolved by software designers. This is not by coincidence if we accept the conjecture that design principles guide us to produce textual representations of the memory network. The existence of analogical structures within the mind would inevitably lead to the evolution of languages and principles that mimic them (although inevitability makes it a no lesser feat by the software designers).

Borrowing heavily from the psychologists definition of the analogy and the software designers interface/class,the structure for the analogy can be represented as in Figure 2. An analogy is the mapping of similar attributes, operations and rules between two or more varieties. Each variety specifies a value for attributes as well as an implementation for the operations. The SaleItem vanilla analogy in Figure 3 serves as a simple example (the N_a, A_a, N_v, A_v annotations are from the analogy template in Figure 2). Of special note is that the

category attribute is an anology itself with varieties "Electricals", "Furnishings" etc. Attributes (A_a) reference another analogy by either the type or the name chosen to represent the attribute (in this case the name "category" is the name of the analogy). The Value (A_v) will reference a variety of that analogy. For our purpose, we view the category analogy as having no attributes or operations and so it is sufficient to reference analogy and variety names only ("category", "Electricals", "Furnishings"). If our purpose changes (either by better understanding or a change in requirement) then we would create an analogy structure for category and reference that in the SaleItem analogy.

The most obvious means to implement analogies in software is to use interfaces and abstract classes but there are other structures that can deliver the same. Please note, the code samples in this section are meant as examples of structures. They show how the code could be written rather than necessarily how they should.

A good guide for identifying analogies in software is if you can "guess" the code that must be written when the behavior is to be extended for a new variety. For example the following code presents the account type attribute for different varieties of futures exchanges (Eurex, Liffe etc)

```
public String getAccountType(String account)
{
    if( exchange.equals("EUREX") )
        return getEurexAccountType(account);
    if( exchange.equals("LIFFE") )
        return getLiffeAccountType(account);
    if( exchange.equals("MONEP") )
        return getMonepAccountType(account);
```

Extending this code for the Matif exchange is easy to "guess":

```
    if( exchange.equals("MATIF") )
        return getMatifAccountType(account);
```

At first view this may seem like an implicit requirement of software development experience. However, the experience required here is one of recognizing analogies, a skill that is developed in our childhood. This implicit assumption of the reader being able to identify and build analogical structures (that reflect business requirements) is arguably a core skill requirement for even a novice software developer.

The next sections detail the various Java code structures that can be used to represent analogies. Naturally, other languages share some of these structures and most have additional ways of coding analogies. I propose that these structures are the templates in the mind that experience builds up. For example, the "Substitute Algorithm" refactor in Fowler's "Refactoring" is replacing the "switch" analogy with an "attribute only" analogy. These structures have evolved using the laws of similarity and proximity to give visual clues to the reader so that they can easily recognize the elements of the analogy.

When coding a solution to a requirement the programmer will use their natural cognitive abilities to identify the analogies of the problem (this will include the simple literal similarities as well as the more complex vanilla analogies and rule analogies). Choosing which code structure to represent the analogies will be dependent upon both the type of the analogy and also the dependencies that the implementations of the varieties require. At the same time the experienced programmer will be aware that the code elements need to be chunked into four elements or fewer.

As our understanding of the requirements change (or even as the requirements themselves change) the choice of code structure for the analogies may need to be revisited. The experienced programmer (with the analogy templates in their mind) finds it easier to visualize alternative analogical structures and so is more likely to deliver a "well designed" refactor in a timely manner.

In some cases we may change classes that have been identified as those that should be closed for modification. This does not mean that our original design failed the open-closed principle. Rather, the new requirements have changed the shape of the analogies of the problem. The previous design was the correct solution for the previous analogical structure, but our new requirements have changed that structure into a different problem. Recognizing the changes to underlying analogies will give comfort to the inexperienced coder that a redesign is the right choice.

2.8.1 Attribute only Analogies

The simplest implementation for attribute only analogies is a map. In this example futures exchanges are seen as literal similarities with the only attribute of interest being the country in which they operate.

```
static Map exchangeCountry = new HashMap();

static {
     exchangeCountry.put("EUREX","Germany");
     exchangeCountry.put("CBOT","US");
     exchangeCountry.put("LIFFE","England");
     //can guess what to do
     //to extend for NYBOT......
}

//usage
public String logMessage
     (String exchangeName)
{
     return "Exchange " + exchangeName +
            " is in " +
```

```
            exchangeCoun-
        try.get(exchangeName);
    }
```

- Analogy Name [N_a] – *exchange* is the prefix of the variable name `exchangeCountry`
- Attribute [A_a] – *country* is the suffix of the variable name `exchangeCountry`
- Variety Name [N_v] – the map key (e.g. `"EUREX"`)
- Value [A_v] – the map value (e.g. `"Germany"`)

2.8.2 Statement Shape Analogies

Here a statement shape is repeated for different fields or elements to provide the implementation for a behavior. The similarity of the statement shape is key for the reader to be able to recognize the analogy (which is to check for a null value then check for a zero value). In this example validating a field is the analogy and each field of the class needs it's own variety of validation.

```
private Date expirationDate;
private Long contractNumber;
private Double price;
private Double quantity;
private String description;

private boolean isValid()
{
    if (expirationDate == null ||
        !( expirationDate.getTime() > 0 ) )
           return false;

    if (contractNumber == null ||
        !(contractNumber.longValue() > 0) )
           return false;

    if (price == null ||
        !( price.doubleValue() > 0 ))
           return false;

    if (quantity == null ||
        !( quantity.doubleValue() > 0 ))
           return false;

    //can guess what the code line is
    //for the String parameter
    //"description" ......

    return true;
}
```

- Analogy Name [N_a] – n/a.
- Operation [O_a] – the method name `isValid` gives the single operation.
- Variety Name [N_v] – the similarity of the statement lines (utilizing the law of similarity) will indicate that the variety is defined by the field name.
- Behaviour [O_v] – the statement structure for each field

2.8.3 Switch Analogy

Similar to the Statement Shape analogy, the implementations are chunked together with a more formal specification of variety type. The example here returns the value for combining two numbers with various operators (add, minus etc.)

```
private int operand;

public double calculate
            (double first, double second)
{
    switch (operand)
    {
           case MULTIPLY:
                  return first * second;
           case DIVIDE:
                  return first / second;
           case ADD :
                  return first + second;
           case SUBTRACT :
                  return first - second;
    }

    return 0;
}
```

- Analogy Name [N_a] – n/a.
- Operation [O_a] – the method name `calculate`.
- Variety Name [N_v] – the case values (e.g. `MULTIPLY`, `DIVIDE` etc.).
- Behaviour [O_v] – the block of the associated case.

2.8.4 Method Name Analogy

Either the suffix or prefix of the method is used to express the analogy. The prefix has the advantage of grouping together the methods when they are listed in alphabetical order (as with many IDEs). The prefix version example is the visitor pattern. The suffix example returns different value types from the `Double` object. This structure is useful when all the varieties share dependencies as these can be chunked within the class.

```
//example 1 - prefix
public void visitExpression(Node a){};
public void visitBlock(Node a){};
```

```
public void visitFile(Node a){};

//example 2 - suffix
Double doubleObj = new Double(0);

double a = doubleObj.doubleValue();
int b = doubleObj.intValue();
long c = doubleObj.longValue();
float d = doubleObj.floatValue();
```

- Analogy Name [N_a] – n/a.
- Operation [O_a] – method prefix or postfix (e.g. `visit`).
- Variety Name [N_v] – remainder of method name (e.g. `Expression, Block` etc.).
- Behaviour [O_v] – method block.

2.8.5 Method Parameter Analogy

The type of parameter passed to a method can distinguish the variety. The example here gives the maximum of two numbers (using the static Math class).

```
float f = Math.max(1.0F, 2.0F);
int i = Math.max(1, 2);
long l = Math.max(1L, 2L);
double d = Math.max(1.0, 2.0);
```

- Analogy Name [N_a] – n/a.
- Operation [O_a] – method name (`max`).
- Variety Name [N_v] – the type of the passed parameter.
- Behaviour [O_v] – method implementation.

2.8.6 Class/Interface Analogy

The class/interface analogy allows for multiple elements collected (chunked) together.

```
abstract class Shape
{
     String name; //square, circle etc
     int numSides;

     abstract double area();

}

class Rectangle extends Shape
{
     double length;
     double width

     double area()
     {
            return (length * width);
     }

}
```

- Analogy Name [N_a] – abstract class name (e.g. `Shape`)
- Attribute [A_a] – fields (e.g. `name`, `numSides`).
- Operation [O_a] – abstract method `area`.
- Variety Name [N_v] – extended class name `Rectangle`.
- Value [A_v] – values of field.
- Behaviour [O_v] – implementation of abstract method (`area` method in `Rectangle`)

2.8.7 Rule Analogy

Generics in Java can be used in the coding of Rule analogies. This can be to identify the operations needed by the rule and also as a way to formally define the type associations (e.g. the object type returned by the Iterator must be the same type as that placed in the Collection class).

2.8.8 Application Level Analogy

The running instances of an application are literal similarities of one another. Configuration files and system properties identify the attributes of the analogy. IOC mechanisms allow us to define operations at the application level analogy.

2.9 Multi-Paradigm Design

The approach of identifying the business analogies in the requirements and choosing the pertinent code structure was identified by Coplien [4]. Coplien uses the term "domain" for analogies and "sub-domains" for their varieties. "Solution domains" are analogy structures in code (although only formal structures such as interface, templates, etc. are identified). The "commonality and variability analysis" is the process of identifying the analogies.

Both Coplien and Booch [1] explain object-oriented design as decomposition attributed to chunking. Structured design is the discipline of chunking but it is my contention that object-oriented design is the discipline of identifying and coding analogies. These are different but complimentary skills that must be employed to produce well designed software.

3. Fundamental Metric of Software Design

"… more what you'd call 'guidelines' than actual rules"
– Capt. Barbossa, Pirates of the Caribbean: The Curse of the Black Pearl

3.1 "4 minus Analogies" Rule

Analogies are the building blocks of cognition. Identifying the analogies of the problem is the first step to the solution. Code structures to represent analogies are part of the experience of the programmer.

Our mind has evolved to manufacture long-term memory networks that are optimized for searching and re-learning costs (as we extend it). The consequence of this is that we are limited to processing four elements or fewer at any time (code elements include statement expression, statement groups, associations, methods, classes, packages etc.). The only exception to this is the number of varieties of an analogy. We can understand a shopping list regardless of its length because we understand that all items need to be treated the same (i.e. find and buy). We will naturally group all the varieties as one element under an analogy description.

Therefore, the core fundamental metric of software design is that software should be chunked in elements of four (or fewer) after allowing for any number of varieties of analogies. The psychologist's definition of a chunk is assumed here so cohesion and coupling considerations are implicit in this rule. This does not mean that a class, say, should only have four methods. Rather, the methods need to be chunked together in groups of four or fewer (again after allowing for any number of varieties of analogies).

Chunking software elements is fairly obvious and is mentioned in many texts. Explaining the permissible exceptions using a simple consistent rule is not. The classic exception is that of a single, simple behavior based on different types, which does not warrant a class structure all to itself. Rather, a simple switch/case statement is more appropriate. As seen in previous sections this is a structure for an analogy and so follows the rule of "4 minus analogies".

Including more than four elements may place a burden on the reader to employ their own chunking strategies on the code. Without the guidelines that the author can give using the structures above, the reader may produce different partitions and/or have difficulty doing so, leading to confusion of the code (and the greater probability of introducing bugs).

Chunking analogies may shed some light on the artistry behind programming but not all. For example Bloch's builder structure owes more to making code look like written language despite the tight restrictions of what has to be a very limited code vocabulary.

4. Example "Companion" Sections

The brief explanations below serve two purposes. Firstly, they provide evidence that the core principles above, which have been extracted from the cognitive model, are consistent with design principles that are widely subscribed to. Secondly, they show how novices can be provided with the details behind when the rules below are appropriate to be applied. For example, the rule of "4 minus analogies" serves as a test for when a method or individual statement is too large or when a class has too many methods etc.

4.1 Design Patterns

The GoF design patterns [13] show how analogies can be delivered using class level structures. Single analogies are discussed in some patterns. Multiple analogies are also discussed, in some cases where one of the analogies is not under direct control (e.g. Adapter).

4.1.1 Abstract Factory

In most cases the elements (typically attributes and behaviors) of an analogy will be chunked together due to the natural strong associations. When there are multiple analogies influencing a behavior, the layering of the analogies will largely be decided by considerations of the dependencies. In the abstract factory pattern each concretion of the WidgetFactory interface creates the varieties of widgets for a single OS. Here, the behaviors of the widgets are subject to two analogies, one that has the widget type as variety, the other has the OS (that the widget is displayed upon) as the variety. An alternative structure may be that the concretions create a single widget for a variety of OS's [in both cases there are (No. Widgets) x (No. OS) classes]. The intent of the pattern is stated as:

"Provide an interface for creating families of related or dependent objects without specifying their concrete classes".

Here the stronger dependencies are stated to be those within the "family" and so the preferred structure groups the variety of widgets for a single OS.

Sometimes, we may need to write the code where the analogies overlap to fully identify the dependencies and so decide which analogy has the stronger associations. If we have prior experience then it may be possible to perform this in the mind and so visualize a design before we put fingers to keyboard. In some cases however, even the most experienced programmer will need to get into writing code before the design structure is finalized (especially if unfamiliar third party code is being used).

4.1.2 State

The state pattern reminds us that the choice of variety within an analogy may be a dynamic one. It also highlights that if the associations amongst the behavior (methods) of each variety are stronger than the associations amongst the varieties (i.e. common code) then it makes sense to chunk the varieties in their own separate classes.

4.1.3 Visitor

The method name analogy as discussed in section 2.8.4.

4.2 Fowler Refactoring

There are a number of common concepts within Martin Fowler's Refactoring book that can be explained with reference to the cognitive model:

a) *Chunking*. Code elements should be chunked in elements of 4 or fewer (allowing for analogies).

b) *Accurate Names*. Names of variables, methods etc serve as images on the LTM network. The more accurate these names are in describing what they represent the more likely that the reader's mind will use the name as an image and so the closer the code structure is to the structure in LTM. As a contrary example, using the same variable name to represent the value throughout the different stages of calculation must mean that it is either inaccurate for at least one of its cases or so vague as to diminish its suitability as an image name.

c) *Introduce Name*. If it is difficult to determine the purpose of the code chunk from the elements within, attaching a name (literal description or metonymy) will serve as an image on the LTM network. In this way the author can efficiently pass knowledge on to the reader.

d) *Remove unnecessary layers of indirection*. Indirections that serve no purpose have an associated cost (see section 2.6 "Code for Experts and Novices").

Example rule explanations are as follows

- *Extract Method* - The use of methods to chunk code including introducing a name.
- *Inline Method* - If the method body is just as clear as the name then not only is this redundant information but it is also an unnecessary indirection.
- *Inline Temp* - Unnecessary indirection
- *Replace Temp With Query* - Reduces the number of elements in the main method to ease cognitive load and also aids chunking.
- *Introduce Explaining Variable* - Chunking and introduces image. A long statement (i.e. one that has greater than four elements) is split into a number of statements that have four or fewer elements, each with their own image (variable name).
- *Split Temporary Variable* - Accurate Names
- *Remove Assignments To Parameters* - Accurate Names
- *Replace Method With Method Object* - Use of a class structure to allow a long method (greater than four elements) to be chunked.
- *Substitute Algorithm* - Swaps one analogy structure for another.

5. Conclusion

This paper has

- Detailed the strong mapping between the cognitive model and design principles.
- Identified that recognising and coding analogies is one of the two primary disciplines in good software design (the other being chunking).
- Used these results to discover the "4 minus analogies" rule.

Making code easier to understand is the primary driver behind the majority of software design principles. To define simplicity, however, we must examine the processes and limits within the mind. The cognitive model can be understood and learned by the novice using their life experiences as examples and does not require any programming or design knowledge. The concepts of cognition are the fundamentals behind design principles. This does not obviate the necessity for learning and understanding design principles but it does help to lower the bar of experience needed for the novice to build up the ability to differentiate when it is appropriate to apply them. The subsequent improvement in the design of our code will reduce the time and costs associated with support and mainentance.

The primary proposal is that novice programmers follow a training program to:

1. Become familiar with the cognitive model, including:
   a) Chunking and the capacity limits of short-term memory.
   b) Analogies as the building blocks of cognition.
   c) Long term memory structures.
   d) Cognitive loading (e.g. split attention effect and redundant information).
   e) Expert versus novice behavior.
2. Identify and write Analogy & Chunking structures in code
3. Understand how texts like Fowler's Refactoring & the GoF Design Patterns provide common solutions to chunking and high level structures of analogies.
4. The importance of learning IDE features which deliver an experts view on code that has been structured primarily for the novice.

6. Signoff

In this paper the analogy is made between the structure of memory within the brain and the structure of code that follows design principles (which have evolved over the last few decades). In this case the base of the analogy is the cognitive model provided by psychologists which when mapped onto the target of design principles, enriches our understanding to lower the bar of experience needed to apply them. At the heart of this mapping are analogies themselves. What if we were to evaluate the analogy in the opposite way? Could the design principles that have surfaced from the melting pot of millions of programmers working on billions of lines of code be used to infer knowledge about cognition?

An analogy that helps us to understand analogies.

Acknowledgments

To Bernie Mullen, Ged Mullen and Pete Williams for the helpful comments and suggestions. Thanks also to Giles Thompson, Miranda Sinclair, John Weir and Don Raab for their support and guidance.

References

[1] G. Booch; R. A. Maksimchuk; M. W. Engle; B. J. Young; J. Conallen; K. A. Houston. Object-Oriented Analysis and Design with Applications, Third Edition (2007). Addison-Wesley ISBN 0-201-89551-X

[2] F. Brooks. No Silver Bullet - Essence and Accidents of Software Engineering (1986). http://www.lips.utexas.edu/ee382c-15005/Readings/Readings1/05-Broo87.pdf.

[3] G. Cooper. Research into Cognitive Load Theory and Instructional Design at UNSW (1998). http://education.arts.unsw.edu.au/staff/sweller/clt/index.html

[4] J. O. Coplien. Multi-Paradigm Design for C++ (2003). Addison-Wesley ISBN 0-201-82467-1.

[5] N. Cowan. The Magical Number 4 in Short-term Memory: A Reconsideration of Mental Storage Capacity. In *Behavioral and Brain Sciences*, Vol. 24, No. 1. (February 2001), pp. 87-185. (2001)

[6] O. Dahl, E. Dijkstra, C. A. R Hoare. Structured Programming (1972). Academic Press.

[7] A. Didierjean, and F. Gobet. Sherlock Holmes – An expert's view of expertise. In *British Journal of Psychology* 99: 109–125 (2087).

[8] D. K. Dirlam. Most efficient chunk sizes. In *Cognitive Psychology*, 3:355–359, 1972.

[9] M. Fowler, K. Beck, J. Brant, and W. Opdyke. Refactoring: Improving the Design of Existing Code (1999). Addison-Wesley ISBN 0-201-48567-2.

[10] D. Gentner. Structure-mapping: A theoretical framework for analogy. In *Cognitive Science*, 7, pp 155-170 (1983).

[11] F. Gobet. Discrimination Nets, Production Systems and Semantic Networks: Elements of a Unified Framework (1996). http://people.brunel.ac.uk/~hsstffg/papers/UnifiedFramework/UnifiedFramework.html.

[12] F. Gobet, P. C. R. Lane, S. Croker, P. C-H. Cheng, G. Jones, I. Oliver, and J. M. Pine. Chunking mechanisms in human learning. In *TRENDS in Cognitive Sciences*, 5, 236-243. (2001).

[13] Gamma, Helm, Johnson and Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. (1995). Addison-Wesley ISBN 0-201-63361-2

[14] Douglas R. Hofstadter. Analogy as the Core of Cognition. In *Dedre Gentner, Keith Holyoak, and Boicho Kokinov (eds.) The Analogical Mind: Perspectives from Cognitive Science, Cambridge, MA: The MIT Press/Bradford Book*, 2001, pp. 499–538.

[15] K. J. Holyoak and P. Thagard. The Analogical Mind (1997). http://cogsci.uwaterloo.ca/Articles/Pages/Analog.Mind.html

[16] J. N. MacGregor. Short-term memory capacity: Limitation or optimization? In *Psychological Review*, 94(1):107–108, 1987.

[17] George A Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. In *The Psychological Review*, 1956, vol. 63, pp. 81-97 (1956)

[18] H. Mills, R. Linger, A. Hevner. Principles of Information System Design and Analysis (1986). Academic Press.

[19] G. Myers. Composite/Structured Design (1978). Van Nostrand Reinhold.

[20] M. Page-Jones. The Practical Guide to Structured Systems Design (1988). Yourdon Press.

[21] T. Stafford, M. Webb. Mind Hacks (2004). O'Reilly ISBN 0-596-00779-5

[22] P. Van den Broek, K. Risden, Y. Tzeng, T. Trabasso, and P. Brasche. Inferential questioning: Effects of comprehension of narrative texts as function of grade and timing. In *Journal of Educational Psychology*, 93(3): 521-529 (2001).

[23] N. Wirth. Program Development by Stepwise Refinement (1983). In *Communications of the ACM* vol. 26(1).

[24] N. Wirth. Algorithms and Data Structures (1986). Prentice-Hall.

[25] E. Yourdon, L. Constantine. Structured Design (1979). Prentice-Hall.