Object-Oriented Patterns: Lessons from Expert Systems

TIM MENZIES
Department of Artificial Intelligence, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2052 `timm@cse.unsw.edu.au`

SUMMARY

Three benefits are typically claimed for object-oriented (OO) patterns: (i) reusing parts of the conceptual models of old implementations; (ii) guiding the current development based using successful previous developments; and (iii) communicating existing systems to newcomers. We will argue that a similar idea can be found in the expert systems literature dating from the early-1980s. The goal of $\mathcal{KL}$ or *knowledge-level* modelling (e.g. KADS) is to identify abstract patterns of inference that appear in many expert systems. Such abstract patterns of inference and program structure, it is argued, are productivity tools for the creation of software applications; i.e. $\mathcal{KL}$ argues for a similar reuse benefit as OO patterns. Recently, however, an alternative view has emerged. While such abstract patterns are good for communications and guidance, the reuse benefits may never be realised. Patterns may be best viewed as tools for structuring an argument, rather than recording a conclusion.

KEY WORDS OO patterns expert systems knowledge-level modelling

INTRODUCTION

An exciting idea in current OO thinking is the *pattern*, i.e. a fragment of a high-level conceptual model which may be useful in many applications. Using patterns may have three benefits:

- **The re-use benefit**: A designer can bootstrap themselves into better systems using proven old systems. Example **reuse** patterns can be found in [1-4]. Note that analysts may not use the patterns verbatim. Reuse patterns are like the logical design which may require some configuration/ alteration for the physical implementation of any particular system. Nevertheless, the essence of the physical implementation will be the reuse pattern.
- **The guidance benefit**: Not all patterns are reusable libraries of OO classes. **Guidance** patterns serve to direct the analyst's focus onto a set of issues that previous analysts have found insightful. Example **guidance** patterns are CHECKS [5] and Caterpillar's Fate [6].
- **The communication benefit**: Patterns are a succinct tool for explaining existing systems. When we tutor OO, we find patterns to be a useful final initiation ritual for a novice OO developer. When they "get" patterns, we know that they are capable of comparing and contrasting a wide range of OO systems.

Despite the current level of enthusiasm for OO reuse patterns, we find it necessary to sound a note of caution. A similar idea, called $\mathcal{KL}$ or *knowledge-level* modelling can be found in the expert systems literature dating from the early-1980s [7, 8]. This paper tries to bridge the gap between abstract conceptual models proposed for OO and abstract conceptual models proposed for expert systems. In all, we will say more about expert systems than OO. In particular, after over a decade of experience with $\mathcal{KL}$, we

CCC 0038–0644/97/010001–01

Received 1 October 1996
Revised 10 May 1996

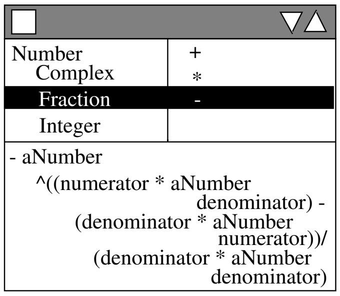

Figure 1. A class hierarchy browser

can make some clear statements about its pitfalls. We will argue that we can extrapolate the lessons of $\mathcal{KL}$ modelling to OO reuse patterns. That is, the problems already seen by $\mathcal{KL}$ will have to be faced in the future by OO pattern researchers. In particular: we doubt the **reuse** benefit but not the **communication** benefit or the **guidance** benefit.

This paper is structured as follows. Due to the hybrid nature of the content of this paper (OO and expert systems), we will first introduce OO design patterns to our expert systems audience. Since our concern is primarily with the **reuse** benefit and not the **guidance** benefit, this section will focus on the GOF/GOV/Fowler/Coad-style reuse design patterns*. Next, we introduce $\mathcal{KL}$ to our software engineering audience and then offer a mapping from OO patterns to $\mathcal{KL}$. Known pitfalls of $\mathcal{KL}$ will be reviewed. Our discussion section offers two action plans based on this revised view of patterns. Roughly speaking, we say that patterns may be best viewed as tools for structuring an argument, rather than recording a conclusion.

Reuse OO Patterns

This section is a short introductory tutorial on OO patterns.

Consider the class hierarchy browser of Figure 1. When a class name is selected in the upper-left list box, the methods of that class are displayed in the upper-right list box. If one of these methods is selected, then the source code for that method is displayed in the bottom text pane.

Now compare this class hierarchy browser with the disk browser shown in Figure 2. When a directory name is selected in the upper-left list box, the files in that directory are displayed in the upper-right list box. If one of these files is selected, then the contents of that file are displayed in the bottom text pane.

Clearly, there is some similarity in the two browsers. Containers (classes or directories) are shown top-left. The things in the containers that are not themselves containers (methods and files) are shown top-right. The contents of these non-container things are shown in the bottom pane.

If we rename containers *composites* and the non-containers *leaves* then we can design one *composite browser* class that handles both class hierarchies and directory trees (see Figure 3). That is, our disk browser and class hierarchy browser are both presentations of nested composites.

Figure 4 shows the inner structure of the composite browser. Composites contain either other composites or leaves. Leaves compile the contents of lower text-pane. Once this structure is in place, all that is required to convert a disk browser to a class hierarchy browser is to:

- Change the title of the window for "Disk Browser" to "Class Hierarchy Browser".

* GOF= the "gang of four" [1]; GOV= the "gang of five" [2]; Fowler [3]; Coad [4]

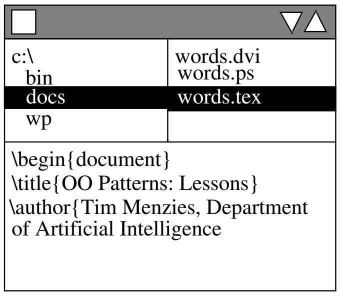

Figure 2. A disk browser

- Define `Class` beneath `Composite` and `Method` beneath `Leaf`.
- Implement the different `compiles` methods in `Method`. `Compiling` file contents implies transferring text to primary storage. `Compiling` method contents implies parsing the source code, etc.

We have just isolated a "pattern": a fragment of a high-level conceptual model which may be useful in many applications. The above design can be used to (i) browsing a disk; (ii) browse a class hierarchy; or, more generally, browse any 1-to-many nested aggregation (e.g. players in teams, persons in companies, stock on shelves). This composite pattern is one of the 23 OO reuse design patterns listed by GOF. The above example suggests the power of such OO reuse patterns. Seemingly different problems can be resolved to a single design. OO reuse patterns could become a repository for experience which can benefit new designers. OO reuse patterns could also serve to unify the terminology of OO design, allowing experience from one application to migrate into another area.

Patterns have been documented in many formats. The GOV prefer the format: *context*, *problem*, *solution* [2]. The context describes a design situation. The problem describes the set of forces that repeatedly occur in that situation while the solution describes a configuration to balance those forces. This solution contains a description of the *static components* and the *runtime behaviour*. In OO patterns: (i) the static components are described using class hierarchies and their relationships; and (ii) the runtime behaviours are described using some variant on collaboration diagrams [9]. The GOV argue that this format of a pattern is compatible with numerous other patterns researchers (page 11 of [2]).

Patterns can be at different layers of abstraction. The GOV describe three layers of pattern ab-

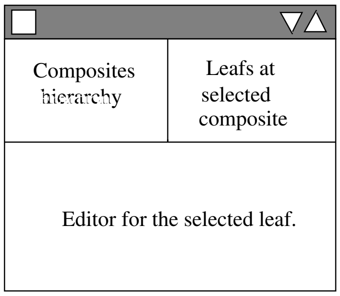

Figure 3. A composite browser

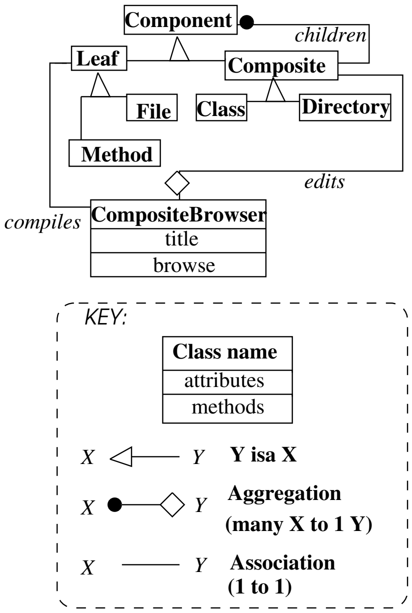

Figure 4. Object model for the composite browser

straction: (i) low-level language-dependent *idiom* patterns; middle-layer language independent *design* patterns describing a programmer's key mechanisms (e.g. the GOF patterns); (iii) and top-level *architectural patterns* that spread across the entire application. Examples of architectural patterns are *layered* architectures (e.g. the three tiered database-model-dialogue systems found commonly in standard management information system-style applications); pipe-and-filter (e.g. the dominant paradigm in UNIX shell scripts); or blackboards (an expert systems technique).

Patterns can be pitched at different audiences. For example, the GOV and GOF patterns are intended for programmers or implementation-aware analysts. Fowler describes *analysis patterns'*; i.e. high-level conceptual patterns which are used to communicate a design to the user community. Fowler was involved in the development of a large medical system. Analysis patterns were used to discuss the design of the system with doctors and nurses. Some patterns found in that medical system (chapter 3 of [3]) were also useful in a corporate finance applications (chapter 4 of [3]).

$\mathcal{KL}$ INFERENCE SKELETONS

Our general claim will be that OO reuse patterns and $\mathcal{KL}$ (e.g. KADS) are similar enough for lessons from $\mathcal{KL}$ to be relevant to OO reuse patterns. This section describes $\mathcal{KL}$. For the moment, we will use the term "inference skeletons" to describe the conceptual models in $\mathcal{KL}$ since we have yet to prove that $\mathcal{KL}$ equals OO patterns (this will be done below).

The goal of $\mathcal{KL}$ modelling is to identify abstract reusable inference skeletons that appear in many expert systems; e.g. diagnosis, classification, monitoring, etc. Such abstract reusable inference skeletons, it is argued, are productivity tools for the creation of expert systems. Examples of $\mathcal{KL}$ are Problem Solving Types [7], Clancey's model construction operators [10], the PROTEGE-II project [11], TINA [12], SPARK/ BURN/ FIREFIGHTER (SBF) [13] and KADS [14]. In terms of mature $\mathcal{KL}$ methodologies,

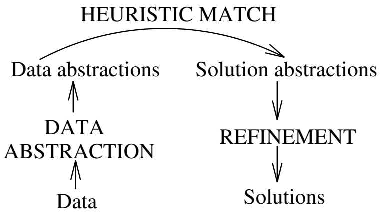

Figure 5. Heuristic classification

KADS is the dominant $\mathcal{KL}$ approach. Wielinga *et. al.* note that, as of 1992, KADS has been used in some 40-to 50 knowledge-based systems (KBS) projects, 17 of which are described in published papers [14].

In his classic *Heuristic Classification* paper, Clancey reverse-engineered 10 expert systems written in a variety of tools and languages. He found that all these systems shared the same abstract inference skeletons, which he called *heuristic classification* (shown in Figure 5).

For example, Figure 6 shows Clancey's analysis of the MYCIN [15, 16] inference structure (abstract schema at top, followed by an example). MYCIN was a backward-chaining rule-based system that prescribed antibiotics. MYCIN worked by building an abstract model of the patient from the patient's data. This is then matched across to a hierarchy of disease classes. The final diagnosis is produced by following the class diseases hierarchy downwards, looking for the most specific disease that is relevant to this patient. Note the similarity with Figure 5.

Figure 7 shows Clancey's analysis of another system, written in LISP which diagnoses an electronic circuit in terms of the component that is causing faulty behaviour. The abstract schema is shown on top, and an example is shown underneath. Note the similarity with Figure 6 and 5.

After the *Heuristic Classification* paper, Clancey refined his inference skeletons. In *Model Construction Operators* [10], Clancey argued that rules like Figure 8 contain domain-specific terminology (see Figure 9) as well as reusable inference strategies (see Figure 10). If these are removed from the rule, then not only have we isolated the true business knowledge in the rule (see Figure 11), but we also have found inference knowledge we can reuse elsewhere. Clancey's preferred architecture for expert systems is (i) a library of pre-defined problems solving strategies such as Figure 10; and (ii) a separate knowledge base containing the special domain heuristics like Figure 11.

Inspired by Clancey's work, subsequent researchers sought other abstract inference skeletons. Tansley & Hayball [17] list over two dozen reusable inference skeletons including systematic diagnosis (localisation and causal tracing), mixed mode diagnosis, verification, correlation, assessment, monitoring, simple classification, heuristic classification, systematic refinement, prediction, prediction of behaviour and values, design (hierarchical and incremental), configuration(simple and incremental) planning, and scheduling. These skeletons are recorded using the KADS notation of Figure 12 in which rectangles are data structures and ovals are functions. Given a `complaint`, the KADS abstract pattern for *diagnosis* is that a `system model` is `decomposed` into `hypothetical` candidate faulty components. A `norm` value is collected from the `system model`. An observation for that candidate is requested from the `observables` (stored internally as a `finding`). The candidate hypothesis is declared to be the diagnosis based on the `difference` between the `norm` value and the `finding`. Note the shaded portions of Figures 12 & 13. We will return to these shaded portions below (see Figure 18).

As another example, Figure 13 shows the KADS abstract inference skeleton for *monitoring*. A `parameter` is selected from a `system model`. The model's expected `normal` value is generated

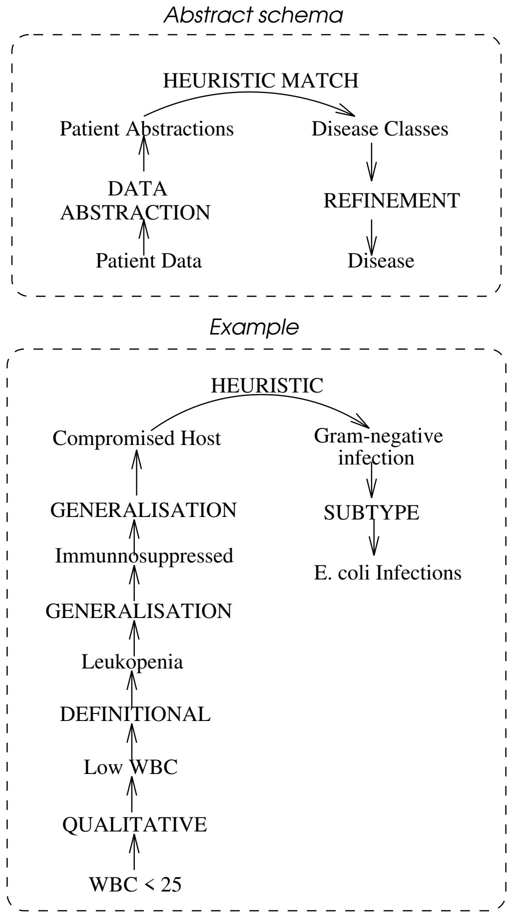

Figure 6. MYCIN

from the model and collected from the `observable-s` (stored as a `finding`). The current state of the monitoring system us reported as a `discrepancy class` after comparing the `finding` with the expected `normal` value.

Note that Figures 5,12, & 13 do not imply a particular execution order of their functions. Conceptually each function can be driven forwards or backwards to connect inputs to outputs or visa versa. The heuristic classification pattern of Figure 5 could be driven from data to solutions to perform diagnosis; i.e. given the `data`, execute forwards `data-abstraction` then `heuristic match`, then `refinement`. Alternatively, it could be driven from solutions to data to perform intelligent data collection; i.e. given `solutions`, execute backwards `refinement`, then `heuristic match`, then `data abstraction`. In this backwards reasoning, the generated `data` items become requests back to the environment in order to rule out certain possibilities. KADS explicitly models this procedural ordering of the function calls in a separate *task layer* diagram.

All known abstract inference patterns skeletons are really combinations of a small number of reusable inference subroutines. Some of the reusable inference subroutines are shown in Figures 12 & 13 (e.g. `select`, `specify`, `compare`). We will see more inference subroutines in the TINA system, below.

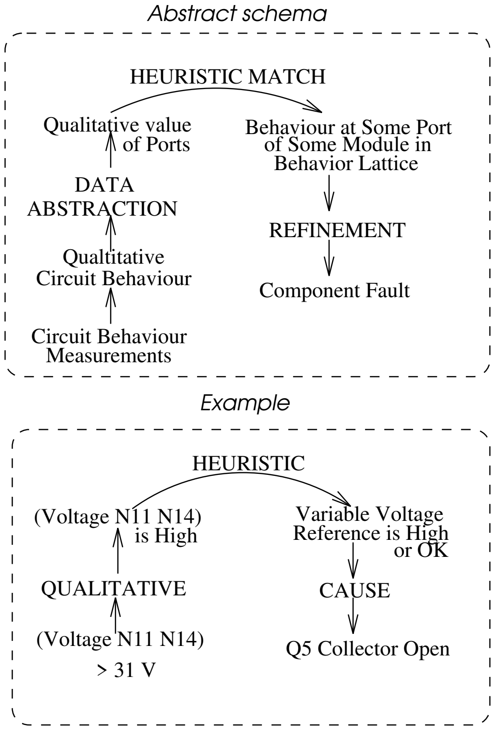

Figure 7. SOPHIE III

New abstract reusable inference skeletons can be quickly built out of these lower-level inference primitives. Libraries of abstract reusable inference skeletons become a productivity tool for building a wide-variety of expert systems. Marques *et. al.* report significantly reduced development times for expert systems using a library of 13 reusable inference subroutines (including `eliminate, schedule, present, monitor, classify, select` and `dialog-mgr`) in their SPARK/ BURN/ FIRE-FIGHTER (SBF) environment. In the nine studied applications, development times changed from one to 17 days (using SBF) compared to a range of 63 to 250 days (without using SBF) [13]. To our knowledge, this is the largest documented evidence of productivity gains in any software approach (be it knowledge based, object-oriented, or otherwise).

```
if   the infection in meningitis                      and
     the type of infection in bacterial               and
     the patient has undergone surgery                and
     the patient has undergone neurosurgery           and
     the neurosurgery-time was less than 2 months ago and
     the patient received a ventricular-urethral-shunt
then infection = e.coli (.8) or klebsiella (.75)
```

Figure 8. A domain rule with hidden reusable inference fragments. From [10].

```
subtype(               meningitis,          bacteriaMenigitis       ).
subtype(               bacteriaMenigitis,   eColi                   ).
subtype(               bacteriaMenigitis,   klebsiella              ).
subsumes(              surgery,             neurosurgery            ).
subsumes(              neurosurgery,        recentNeurosurgery      ).
subsumes(              recentNeurosurgery,  ventricularUrethralShunt).
causalEvidence(        bacteriaMenigitis,   exposure                ).
circumstantialEvidence( bacteriaMenigitis,  neurosurgery            ).
```

Figure 9. Domain-specific terms from Figure 8.

The above description is only a partial description of $\mathcal{KL}$ in general and KADS in particular. For an on-line versions of the KADS documentation, see `http://www.swi.psy.uva.nl/projects/CommonKADS/Reports.html` and `http://swi.psy.uva.nl/projects/CommonKADS/Papers.html`. See [17] for a detailed text on KADS. See the *Related Work* section of [14] for a discussion of the differences in the various $\mathcal{KL}$ approaches. For a tutorial introduction to KADS, see [18, 8]. For an advanced use of KADS-type models, see the TINA system (below).

$\mathcal{KL}$ = OO Patterns

This section argues that it is inappropriate to declare $\mathcal{KL}$ inference skeletons to be different from OO patterns based on their observed notational differences. The intention of the $\mathcal{KL}$ researchers is the same as the OO patterns. Inference patterns satisfy the GOV definition of a pattern; i.e. they have context, problem, and solution. Hence, we argue that $\mathcal{KL}$ inference skeletons are essentially the same as OO patterns. Further, we will demonstrate that the $\mathcal{KL}$ patters are, in some respects, better patterns than the OO patterns.

Are the Notational Differences Significant?

OO patterns are usually expressed in a different notation to $\mathcal{KL}$ inference skeletons (compare Figure 4 with Figure 12). However, just because $\mathcal{KL}$ inference skeletons are not expressed in an OO format, that does not mean they aren't patterns:

- OO patterns researchers agree that patterns need not be expressed only as networks of classes (pages 23-24 of [2]). For example, Coplien refers to the 150 patterns gathered at Bell Labs for telecommunication systems. He remarks that "none are really object-oriented" [19].

Strategy	*Description*
exploreAndRefine	Explore super-types before sub-types.
findOut	If an hypothesis is subsumed by other findings which are not present in this case then that hypothesis is wrong.
testHypothesis	Test causal connections before mere circumstantial evidence.

Figure 10. Problem solving strategies from Figure 8.

```
if   the patient received a ventricular-urethral-shunt
then infection = e.coli (.8) or klebsiella (.75)
```

Figure 11. The business knowledge of Figure 8.

- Patterns can be found in the data-modelling world (chapter 4 of [3]) where they are expressed in an entity-relationship format.
- Patterns were first described by Alexander as a tool for architectural design [20]; i.e. originally, patterns were expressed in a non-programming format.
- Shaw & Garlan's text discusses common software architectures [21] using a free text format. The GOV's section on architectural patterns translates the Shaw & Garlan, into an OO notation under the headings "context-problem-solution". We would argue that the GOV translation did not add significantly to the Shaw & Garlan material. That is, Shaw & Garlan were discussing "patterns"; they just structured their material in a different manner.

Another reason to reject $\mathcal{KL}$ inference skeletons as being true patterns is that they are not described in the way proposed by the GOV; i.e. context, problem, and solution. Such a rejection may not be justified:

- We will argue below that tacit in the $\mathcal{KL}$ inference skeletons is a very strong notion of context,

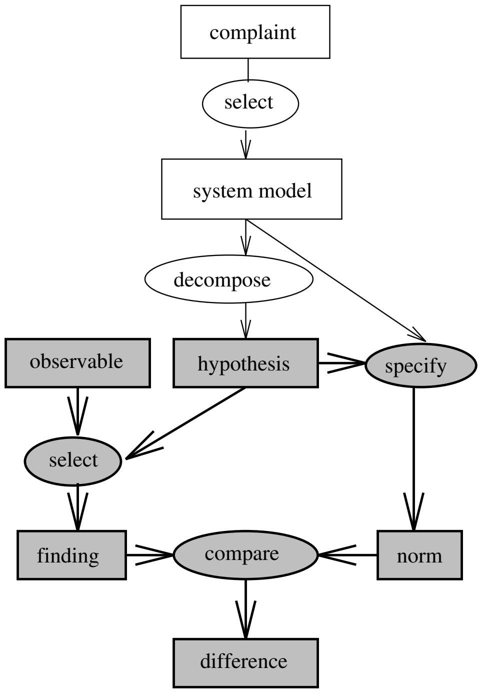

Figure 12. KADS: diagnosis

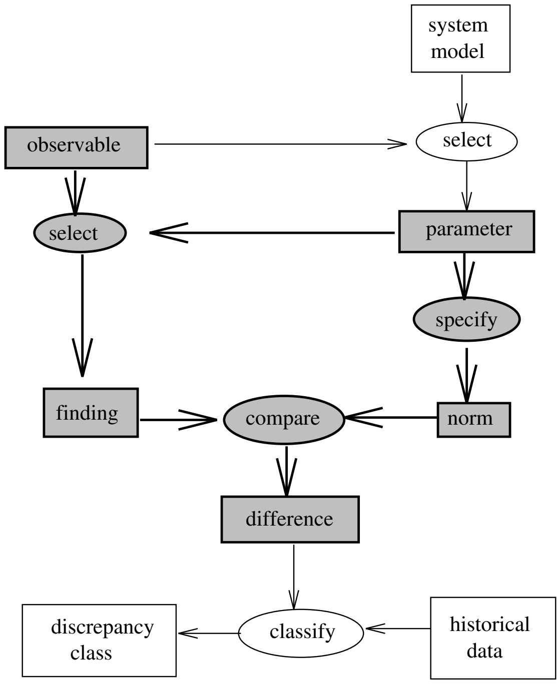

Figure 13. KADS: monitoring

problem, and solution.

- Several prominent patterns texts do not use the context/ problem/ solution format explicitly (i.e. Fowler [3] and Coad [4]), yet clearly represent patterns research. For example, several of the patterns found in Coad can also be found in the GOF text, with small changes. Also, the foreword of the Fowler book is an enthusiastic patterns-based endorsement by one of the GOF authors.

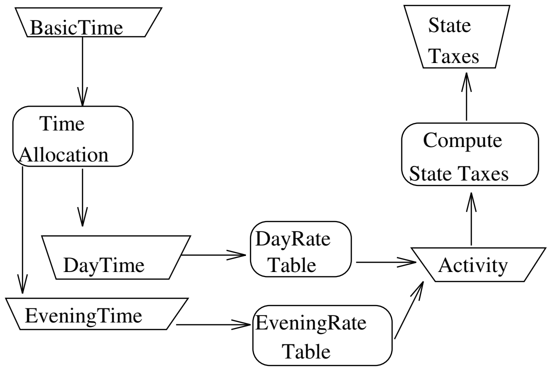

Figure 14. Fowler's process modelling notation; from page 153 of [3].

Interestingly, sometimes an "OO pattern" can look a lot like a "$\mathcal{KL}$ inference skeleton". For example, consider Fowler's proposed high-level notation for dynamic events Figure 14. Fowler notes that this notation is experimental, but stresses the importance of a high-level functional view like Figure 14. In Fowler's proposed notation, ovals are event-driven controllers or triggers on tables which, when required, will call methods inside the classes (polygons). For example, when `day time` writes to `day rate table`, the triggers in that table will call methods within the `activity` class. If we (i) divide Fowler's class polygons into one rectangle for each public method of each class, and (ii) replace Fowler's ovals with ellipses, then the Fowler diagram become very similar to the KADS inference skeletons shown in Figures 12 & 13.

Goals & Uses

Notational comparisons aside, the intent of the $\mathcal{KL}$ modelers and the OO patterns researchers is clearly the same. Consider the quote Clancey used to open the classic $\mathcal{KL}$ paper *Heuristic Classification* [22]. We believe that this quote reflects something of the same intent as the OO patterns community:

> To understand something as a specific instance of a more general case- which is what understanding a more fundamental principle of structure means- is to have learned not only a specific thing but also a model for understanding other things like it that one may encounter. *J.S. Bruner*

Both $\mathcal{KL}$ and OO patterns are examples of the same software abstraction process. Experienced software engineers can describe different applications using a common, abstract, language. Both $\mathcal{KL}$ and OO design languages seek some characterisation of a design that is implementation independent and re-usable. The $\mathcal{KL}$ literature shows that they seek to use $\mathcal{KL}$ inference skeletons in a similar manner to OO patterns:

- The SHELLEY [14] workbench project seeks to gain from the **reuse benefit**. Using SHELLEY, knowledge engineering becomes a structured search for an appropriate inference pattern. Knowledge engineers search requirement documents for a match between the stated requirements and the library of known abstract reusable inference patterns. Once such a pattern is found or developed, then systems development becomes a process of filling in the details required to implement that abstract inference pattern. productivity tools for new applications.
- Numerous $\mathcal{KL}$ researchers argue for the **communication benefit**. For example, practioners find this retrospective second-glance at their systems useful for developing more generalised architectures for future work [18]. Expert systems theoreticians have used $\mathcal{KL}$ to assess and clarify the essential features and differences of applications [23]. Lastly, knowledge engineering novices can use a $\mathcal{KL}$ analysis of classic expert systems to quickly review successful techniques.
- However, to our knowledge, $\mathcal{KL}$ researchers do not argue for the **guidance benefit**.

Structure

To demonstrate conclusively that our $\mathcal{KL}$ inference skeletons are the same as the patterns seen in the OO world, we must show that $\mathcal{KL}$ inference skeletons match the GOV definition; i.e. context, problem/solution, and descriptions of static components with their runtime behaviour. This section will argue that not only do $\mathcal{KL}$ inference skeletons contain "context-problem-solution", but the understanding of the mapping between problems and solutions is more advanced in $\mathcal{KL}$ than in OO patterns.

Context

Clearly, all the $\mathcal{KL}$ inference skeletons have a context. The contexts of the Tansley & Hayball work were listed above and included systematic diagnosis (localisation and causal tracing), mixed mode diagnosis, verification, correlation, assessment, monitoring, simple classification, heuristic classification, systematic refinement, prediction, prediction of behaviour and values, design (hierarchical and incremental), configuration(simple and incremental) planning, and scheduling.

Problems & Solutions

$\mathcal{KL}$ can offer detailed descriptions of problems and their mapping to solutions. For example:

- The TINA [12, 24] system offers a 45 word language for describing problem constraints divided up into "epistemological criteria" (e.g. `fault_behaviour_ not_constrained`), "environmental criteria" (e.g. `imprecise_values`). and "assumption criteria" (e.g. `single_fault_assumption`).
- SBF [13] allows users to graphically draw a network reflecting their business process.

The problems/solutions component of the GOV definition is often expressed as an `IF-THEN` rule. In some cases, the rules describing the mapping between problems and solutions in $\mathcal{KL}$ libraries is so well understood that they can be directly executed. The result can be an automatically configured system. For example:

- SBF can automatically map its business function graphs into its library of inference sub-routines. Once this mapping has been made, a rule-base can be generate which solves the business problem. SBF worked in the context of configuration.
- A similar tool for automatically generating a runtime solution from a high-level problem description can be found in Protege-II [11]. Protege-II is intended to be a multi-context tool but most of its published applications have been in the area of skeletal plan refinement (instantiating a general plan to a particular circumstance) or propose-and-revise (proposing an initial design, then modifying inappropriate portions of that design).
- Benjamin describes TINA, an automatic configuration device for configuring solutions to different problems in the context of diagnosis [12, 24].

TINA was a "proof-of-concept" prototype only and is not as sophisticated as SBF or Protege-II. However, the TINA technique is quite succinct. We will use this system to demonstrate how $\mathcal{KL}$ researchers automatically map problem statements into solutions. In TINA, a solution is a *problem solving method* (PSM) which must be configured for a particular *sub-context* using a set of *primitive inference techniques*. For example:

- Sub-contexts of diagnosis are defined by constraints within the domain such as the availability or absence of `simulation rules`.
- A primitive inference within `prediction based filtering` could be a set `intersection` sub-routine.

A TINA problem is described via *suitability criteria* categorised into a small number of *types*. For example `inference_rules` and `simulation_rules` are suitability criteria with the same type of `constraint_suspension_ method`. The TINA system can automatically reflect over a set of rules describing the transformation process from problems to solutions (or, in the language of TINA, types of suitability criteria into PSMs). A simplified version of some of TINA's rules is given in Figure 15. Types of suitability criteria are shown in the `when` sections. When executing a `then` section,

```
 1 diagnosis
 2 if   prime_diagnostic_method
 3 then symptom_detection           and
 4      hypothesis_generation       and
 5      hypothesis_discrimination.
 6
 7 symptom_detection
 8 when ask_user_method
 9 then apply_user_judgment.
10
11 symptom_detection
12 when compare_symptom             or
13      detection_method
14 then generate_expectation        and
15      compare.
16
17 hypothesis_generation
18 when empirical_hypothesis_
          generation_method
19 then associate                   and
20      prediction_filter.
21
22 hypothesis_generation
23 when model_based_hypothesis_method
24 then find_contributors           and
25      transform_to_hypothesis_set and
26      prediction_based_filtering.
27
28 hypothesis_generation
29 when hypothesis_generation_method
30 then select_hypothesis           and
31      collect_data                and
32      interpret_data.
```

Figure 15. Portions of the TINA rules used for converting problems descriptions into solutions. Adapted from [12].

.

if the PSM component is named in another rule, the reflection can recurse. For example, executing line 3 `symptom_detection` makes TINA test the suitability of the rules at lines 7 and 11.

A sample fragment of TINA output is shown in Figure 16. The `trace_back_method` traces back the dependents of the broken component to find potential contributors to the fault. In the case of multiple contributors, TINA is saying that in this sub-context, they can be simply `intersected`. The resulting contributors set is assessed using the `corroboration method`. Innocent contributors are `deleted` (innocence is computed via running a high-level simulation). The remaining contributors are potentially guilty of the faults and another sub-routine is called to discriminate between them. Note that this `corroboration_method` was generated when TINA explored `prediction_based_filtering` on line 26 of Figure 15 (using rules not shown in this article). For full details of this example, see [12].

Static & Runtime Behaviour

One difference between $\mathcal{KL}$ patterns and OO patterns is their different emphasis of descriptions of static components vs runtime behaviours. As we move from the GOF to the GOV to Fowler, we see a common approach to static structure descriptions (OO notations) and an increasing focus on describing

```
model_based_hypothesis_generation_method {
   trace_back_method;
   intersection_method;
   corroboration                        }

trace_back_method {
   find_upstream  }

intersection_method {
   intersection     }

corroboration_method {
   select_random;
   simulate_hypothesis;
   compare;
   delete           }
```

Figure 16. After exploring its problems/solution mappings, TINA can automatically generate a PSM for diagnosis. Adapted from [12] and converted into a procedural formalism.

the runtime behaviour. $\mathcal{KL}$ patterns offer a very rich description of the runtime behaviour but are less focused on the static descriptions. The $\mathcal{KL}$ patterns we have seen to date are modelled in a functional decomposition style. Hence:

- Unlike OO, $\mathcal{KL}$ operations are not modelled with data structures. Rather, they are modelled separately as inference sub-routines like `classify` (Figure 12) and `select_random` (Figure 16).
- KADS does not show the relationships between their entities. The only "relationships" modelled are those representing data flows between functions.

If "patterns" was purely an OO concept, then this functional decomposition approach to the specification of $\mathcal{KL}$ inference skeletons would disqualify them as patterns. However, we argued above that the concept of a pattern transcends the OO paradigm.

$\mathcal{KL}$ Patterns > OO Patterns?

OO patterns dates back to the early 1990s [25] while $\mathcal{KL}$ dates back to nearly a decade earlier [7]. $\mathcal{KL}$ patterns research is more advanced than OO patterns research:

- $\mathcal{KL}$ patterns have been refined to the point where libraries of pattern solution components can be automatically configured into executable solutions from a problem description.
- Librarian software has been built to support the intelligent indexing of $\mathcal{KL}$ patterns. For example, if TINA finds that its problem cannot be solved using its known solutions, it conducts a dialogue with the user about which modelling assumptions can be relaxed. Further, a programmer can go to TINA with only a partial description of their proposed solution, and TINA will fill in the rest after a search for patterns related to that description.

Model	% disorders identified	% knowledge fragments identified
1: epistemological	50	28
2: KADS	55	34
3: no model	75	41

Figure 17. Analysis via different models

LIMITS TO $\mathcal{KL}$

We have made our case that $\mathcal{KL}$ research is essentially the same as OO patterns excepting that the former uses a functional decomposition notation while that latter uses an OO notation. This section discusses the significance of that equivalence. What can OO patterns research learn from the $\mathcal{KL}$ work?

On the positive side, OO patterns researchers can import a large, well-defined body of knowledge about patterns in inference. OO patterns are mostly specified via a description of their static components. $\mathcal{KL}$ patterns are specified via a description of their runtime components. OO patterns researchers could learn some useful tricks about runtime behaviour from their $\mathcal{KL}$ counterparts.

On the negative side, even after a decade of research, the productivity benefits of patterns has yet to be conclusively demonstrated in the $\mathcal{KL}$ field. The rest of this section discusses these productivity issues. Clancey's *Heuristic Classification* paper [22] offered a unified retrospective view on numerous, seemingly different, expert systems. Similar (but smaller) studies (e.g. [26, 18]) suggest that $\mathcal{KL}$ can retrospectively clarify historical expert systems design issue. However several important productivity issues remain outstanding: do $\mathcal{KL}$ abstraction assist in the design of new systems?; are the $\mathcal{KL}$ abstractions re-usable?; and does the extra level of abstractions used in $\mathcal{KL}$ overly-complicate the design process?. This questions are discussed below.

Do $\mathcal{KL}$ Abstractions Assist in the Design of New Systems?

Corbridge *et. al.* reports a study in which subjects had to extract knowledge from an expert dialogue using a variety of abstract pattern tools [27]. In that study, subjects were supplied with transcripts of a doctor interviewing a patient. From the transcripts, it was possible to extract 20 respiratory disorders and a total of 304 "knowledge fragments" (e.g. identification of routine tests, non-routine tests, relevant parameters, or complaints).

Subjects were also supplied with one of three abstract reusable patterns representing models of the diagnostic domain. Each model began with the line "To help you with the task of editing the transcript, here is a model describing a way of classifying knowledge". Model one was an "epistemological model" that divided knowledge into various control levels of the diagnosis process. Model one was the "straw man"; it was such a vague description of how to do analysis that it should have proved useless. Model two was a more sophisticated version of Figure 12. Model three was "no model"; i.e. no guidance was given to subjects as to how to structure their model. The results are shown in Figure 17.

The statistical analysis performed by Corbridge *et. al.* found a significant difference between the performance of groups 3 compared to groups 1 and 2. No significant difference could be found between the poor-abstract-model group (model 1) and the group that was using a very mature abstract model (model 2). These are very counter-intuitive results. Using a hastily-built abstraction was just as useful as using a mature abstraction. And using no abstractions worked best of all! Far from challenging this result, the $\mathcal{KL}$ community is now exploring empirical methods for exploring its approach in the

Sisyphus-3 project*.

Are $\mathcal{KL}$ Abstractions Re-usable?

It is not clear that the problem solving methods found by $\mathcal{KL}$ are truly reusable. Often, researchers build their own preferred patterns rather than use existing ones. The Tansley & Hayball patterns [17] are very different to the patterns offered by other researchers; e.g. Bredeweg's qualitative prediction method [29]. In the literature already reviewed by this paper, we can find four different versions of diagnosis:

- Benjamin's TINA's view,
- KADS (Figure 12),
- Heuristic classification (Figure 6, & 7),
- Tansley & Hayball's approach [17].

To this list, we can add numerous other approaches to diagnosis:

- An OO-based approach described in chapter 3 of Fowler [3].
- Our own graph-theoretic approach [30].
- DeKleer & William's approach based around a distinction between a problem solver and a assumption-based truth maintenance system [31].
- Poole's approach based on Bayesian reasoning [32].
- To name but a few (for more examples, see [33]).

While some of the these patterns share some common features, they reflect fundamentally divergent different views on how to perform diagnosis. For example, like DeKleer, we view assumption space management as the key inference strategy within diagnosis [30]. In assumption space management, mutually exclusive assumptions are managed in separate logical worlds. An expert system can reflect over that assumption space to intelligently select the next test to perform (a "good" test costs very little and culls much of the assumptions space). The implementation of such a multiple-worlds approach is a non-trivial task. It adds significant amounts of extra architecture to the implementation. This approach is not explored by Fowler, KADS, or Tansley & Hayball. We therefore note that, at least in the case of diagnosis:

- The pattern has not stabilised with time;
- The pattern may not do so in the foreseeable future.

More generally, between the various camps of $\mathcal{KL}$ researchers, there is little agreement on the details of the inference patterns. The list of inference sub-routines from KADS [14] and SBF [13] are significantly different. Also, the number and nature of the problem solving methods is not fixed. Often when a domain is analysed using $\mathcal{KL}$ a new method is induced [18]. In summary, since inference patterns have not stabilised over time, then extensive **reuse** is unlikely.

Does $\mathcal{KL}$ Modelling Overly-Complicated Modelling?

Certain $\mathcal{KL}$ authors note certain similarities between the different abstract reusable inference patterns proposed by KADS:

* Sisyphus is an attempt by the international KA community to define reproducible KA experiments [28].

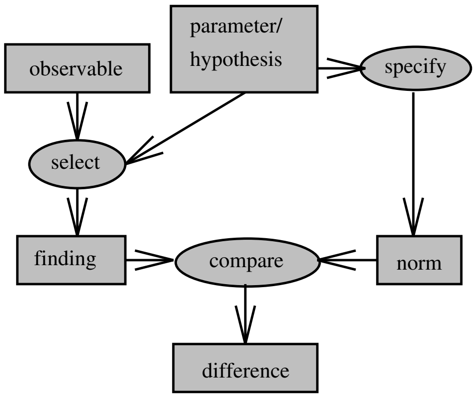

Figure 18. Overlap between KADS diagnosis and monitoring

- Scheduling, planning and configuration are actually the same problem, divided on two dimensions ("goal states known or not" and "temporal factors considered or not" (Figure 12.3 of [17]).
- There exists a common sub-graph between Figures 12 & 13 (see Figure 18).

Having noted that such similarities exist, the $\mathcal{KL}$ researchers do not take the next step and simplify their distinctions (e.g. by combining diagnosis and monitoring). In other work [30, 34], we have explored unifying knowledge-based processing by inferencing over and-or graphs. Such graphs could be computed from an OO design if we ignore all encapsulation boundaries and just map the dependencies between instance variables. In terms of abstraction, we would characterise such a modelling technique as a very-low level tool. Nevertheless, we have found that a single inference procedure (abduction) can be applied over such a representation to implement many of the $\mathcal{KL}$ abstract reusable inference patterns (e.g. diagnosis, case-based reasoning, explanation, prediction, prediction, classification, planning, monitoring, qualitative reasoning, verification, multiple-expert knowledge acquisition, explanation, single-user decision support systems, multiple-user decision support systems, natural-language processing, design, visual pattern recognition, analogical reasoning, financial reasoning, machine learning, and case-based reasoning).

Our general point here is that, in the case of $\mathcal{KL}$, abstractions have *confused* rather than *clarified* the modelling process. In this regard, the analysis of Motta & Zdrahal of the Sisyphus-2 applications to be particularly interesting. Motta & Zdrahal discuss the various Sisyphus-2 $\mathcal{KL}$ implementations using their special knowledge of constraint satisfaction algorithms [35]. We find their lower level more insightful into the construction process than the less-detailed, high-level $\mathcal{KL}$ approach. This low-level view of a problem can find errors that experienced $\mathcal{KL}$ practioners cannot. For example, Motta & Zdrahal argue that one declarative translation of the procedures in the Sisyphus-2 specification blurred the distinction between hard constraints (which must not be violated) and soft constraints (which can be optionally violated) [36].

DISCUSSION

Based on our experience with an equivalent line of research in expert systems, we have argued that OO patterns are not a productive **reuse** tool. This is a counter-intuitive position. The overwhelming intuition is that the abstractions offered by OO reuse design patterns and $\mathcal{KL}$ are true and useful for the construction of new systems. Ralph Johnson comments:

> The reason that I believe that OO design patterns are reusable is because I reuse them. The reason I believe they help people design is because people who learn them tell me that they help them design. The reason that I am *sure* that different groups of designers use different patterns is because I've seen it happen [37].

This quote sounds a lot like something from the KADS community; i.e. abstract descriptions of old designs are a penetrating and useful insight into the design process. However, after a decade of $\mathcal{KL}$ research, we now doubt this insight, at least for $\mathcal{KL}$:

- Abstract patterns developed by different developers can be different.
- The number of abstract patterns seems unbounded; practioners keep inventing new one.
- Practioners don't reuse each others' supposedly reusable abstractions.
- When we actually experiment with supposedly reusable patterns and productivity, (e.g. the Corbridge study) we see evidence to support the counter-intuitive conclusion that well-formed mature supposedly reusable patterns are *less* productive than no pattern at all.

So, what is the appropriate use of the reuse patterns offered by (e.g.) GOF, GOV, Fowler, KADS, etc? We make two suggestions. Firstly, we should monitor OO patterns for $\mathcal{KL}$ -type problems. Secondly, we not use them as objective canonical versions of truth, but as an assistant in analysis and design.

Monitoring Potential Problems with Patterns Reuse

We hope we have, at the very least, motivated the need for experimentation to test if patterns are indeed reusable. This section describes a series of such tests,

If, in the year 2007 we see the following, then we can conclude that the **reuse** benefits for OO patterns are illusionary:

- Between different OO "reusable" patterns, common processing elements are identified suggesting that seemingly different "reusable" patterns have a significant overlap.
- The extra layer of abstraction used in OO "reusable" patterns confuses rather than clarifies design issues.
- When controlled experiments are performed measuring development productivity, then no difference is detected between those applications that use and do not use OO "reusable" design patterns.
- In the future, we find that low-level details (e.g. constraint-satisfaction algorithms or and-or graph processing) become the focus of the design process.
- OO "reusable" patterns prove not to be reusable.
- Different developers working on the same problem develop different "reusable" patterns.
- For the above two reasons, libraries of "reusable" design patterns from different sources contain significant deviations.

On the last point, we note that we can already see differences of opinion of the exact nature of the patterns:

- The GOV's design-level patterns are different to those of the GOF (page 380 of [2]);
- There exists a non-trivial overlap between 3 of the GOV patterns: MVC (page 125 of [2]), View Handler (page 291 of [2]), and Publisher/Subscribe (page 239 of [2]). We would propose to combine them.

- We place a very different emphasis to Fowler on graph-theoretic designs. Fowler offers 1.5 pages on "plans and protocols as graphs"(pages 166-68 of [3]). We have argued above that generalised and-or graphs are a engineering framework useful in numerous contexts. For example, recall TINA's search for candidates that could contribute to the current diagnostic problem (Figure 16). Such a search is greatly simplified if the program can access the dependency graph between business concepts. So, in our graph-theoretic view, we would propose to replace much of the diagnostic architecture of Fowler chapters 3 & 4 with a more intricate version of "plans and protocols as graphs".

With respect to these last two points, it is a debatable point whether or not our design proposals are better than those given by the GOV or Fowler. We argue below that it is important that such debates take place, in public, and in front of students of OO design.

Constructivist Patterns

Having raised some doubts over patterns, we still believe patterns are useful. When training OO practioners, we have observed that patterns are a powerful tool for communicating the insights of experienced designers to less-experienced designers. However, it may be a mistake to assume that just because students study patterns, that they will actually use them in their own work. This section uses some theory from computer-based education to claim that while patterns may not be useful as objective records of canonical truth, they may be very useful in supporting analysts and designers as they construct their systems. That is, while we doubt the **reuse** benefit, we are encouraged by the **guidance** benefit of patterns. Patterns may be best viewed as tools for structuring an argument, rather than recording a conclusion.

Objectivism vs Constructivism

Education theorists are turning away from *objectivist* and towards *constructivist* approaches to education [38]. In objectivism, the teacher presents canonical forms of truth to students. In constructivism, the teacher presents a range of different viewpoints on the one issue to students. The training session and the students' role is different in both approaches. In objectivism, students must memorise facts or models which they will apply later. In constructivism, (i) teachers support students as they explore a debate over some issue; and (ii) students try to build their own views of best practice.

An objectivist teacher may present some impressive theoretical mental framework summarising their own view of a field. This summary may have taken years to synthesis. An constructivist teacher knows that while such summaries marked a significant watershed in their own understanding of a field, they may not be the final goal for the students. Rather, such theoretical frameworks may be useful to students only as guides to reviewing a field while each student constructs their own personal mental frameworks.

Our reading of the current $\mathcal{KL}$ and OO patterns literature is that it is tacitly objectivist: i.e. the libraries are assumed to be canonical forms of best solutions and a source of power in their own right. The constructivist process of using a pattern to explore options has not been a major focus of the current OO or $\mathcal{KL}$ patterns literature.

A Constructivist use of OO Patterns

A hypothetical constructivist patterns toolkit would support the following features:

- Practioners could browse a library of $\mathcal{KL}$ or OO solutions to the same problem. Instead of reading up on "one" solution, a constructivist patterns text would present multiple solutions to

(e.g.) the diagnostic problem (as done above when we discussed *are $\mathcal{KL}$ abstractions reusable?*) or the update problem between multiple views. Techniques from case-based reasoning may be relevant here [39].
- Each alternative solution to a problem is annotated with its strengths and weaknesses. Within the OO world, the Fowler, Coad, and Riel [40] texts are a first step in this direction. Fowler presents his patterns in an evolutionary form. From some similar initial design, Fowler takes the reader through a discussion of when the current form would suffice and when it should be made more intricate. Coad takes a simpler "extend, discuss, refine" approach, but in greater detail. Riel presents small designs (less than ten classes) along with a dialogue describing the strengths and weaknesses of those design. He then makes some preliminary remarks about patterns in design modifications which he calls *transformational design patterns* (chapter 10 of [40]).
- Practioners can try parts of those models on their own problem and when they do, some tool encourages them to explore the strengths and weaknesses of that approach. Exactly how this is done is an open research issue. Riel's design heuristics might be suitable for small-designs (though they are far from being universally accepted). However, heuristics for assessing larger-scale 00 architectures or alternative $\mathcal{KL}$ inference skeletons are still poorly documented.

Interim Advice on Using Patterns

The open issues in the last two points of the previous section could be the basis for an research programme that could take many years to terminate. Practioners cannot wait on the outcomes of such a programme. In the interim, we offer the following advice on how to use the current generation of OO or $\mathcal{KL}$ patterns in a constructivist manner:

- Seek alternative patterns for the same context.
- When studying the alternatives, form a group and argue each alternative with respect to some concrete design problem.
- Record not only the resolution of the debate, but the steps in the argument that resulted in that resolution.
- Heuristics suitable for your local institution will be found within those argument steps. Whenever options are being considered, try and discover how your group explores and evaluates options.
- Seek techniques for optimising this exploration/evaluation process.

We speculate that these locally-generated heuristics for reviewing a design may become a tool at least as powerful as reading patterns.

CONCLUSION

We have distinguished three potential benefits from OO patterns: **reuse, guidance**, and **communication**. Based on our experience with similar patterns research in expert systems, we doubt the **reuse** benefit, but endorse the **communication** and **guidance** benefit. The real power of patterns is how they encourage practioners to succinctly describe and review the essence of complex architectures. Patterns should not be viewed as objective canonical knowledge, but tools for supporting a constructivist approach to analysis and design.

ACKNOWLEDGEMENTS

Ralph Johnson's enthusiasm for exploring an alternative view made this paper possible. Norm Kerth pointed out the **guidance** benefit of patterns. The comments of the anonymous reviewers prompted the

creation of half of the discussion section and a significant clarification of our mapping from $\mathcal{KL}$ to OO patterns.

REFERENCES

1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
2. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *A System of Patterns: Pattern-Oriented Software Architecture*, John Wiley & Sons, 1996.
3. M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison Wesley, 1997.
4. P. Coad, D. North, and M. Mayfield, *Object Models: Strategies, Patterns, and Applications*, Prentice Hall, 1997.
5. W. Cunningham, 'The checks pattern language of information integrity', J. Coplien and D. Schmidt (eds.), *Pattern Languages of Program Design*. Addison-Wesley, 1995. Also available at *http://c2.com/ppr/checks.html*.
6. N. Kerth, 'Caterpillar's fate: A pattern language for transformation from analysis to design', J. Coplien and D. Schmidt (eds.), *Pattern Languages of Program Design*. Addison-Wesley, 1995. Also available from *http://c2.com/ppr/catsfate.html*.
7. B. Chandrasekaran, 'Towards a Taxonomy of Problem Solving Types', *AI Magazine*, 9–17 (1983).
8. T.J. Menzies, 'Limits to Knowledge Level-B Modeling (and KADS)', *Proceedings of AI '95, Australia*. World-Scientific, 1995.
9. G. Booch, I. Jacobsen, and J. Rumbaugh, *Version 1.0 of the Unified Modeling Language*, Rational, 1997. http://www.rational.com/ot/uml/1.0/index.html.
10. W.J. Clancey, 'Model Construction Operators', *Artificial Intelligence*, **53**, 1–115 (1992).
11. T.E. Rothenfluh, J.H. Gennari, H. Erikson, A.R. Puetra, W. Tu, and M.A. Musen, 'Reusable ontologies, knowledge-acquisition tools and performance systems: Protege-II solutions to sisyphus-2', B.R. Gaines and M. Musen (eds.), *Proceedings of the 8th AAAI-Sponsored Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, 1994, pp. 43.1–43.30.
12. R. Benjamins, 'On a role of probem solving methods in knowledge acquisition- experiments with diagnostic strategies', *Proceedings of the European Knowledge Acquisition Workshop, 1994*, 1994.
13. D. Marques, G. Dallemagne, G. Kliner, J. McDermott, and D. Tung, 'Easy Programming: Empowering People to Build Their own Applications', *IEEE Expert*, 16–29 (1992).
14. B.J. Wielinga, A.T. Schreiber, and J.A. Breuker, 'KADS: a Modeling Approach to Knowledge Engineering.', *Knowledge Acquisition*, **4**, 1–162 (1992).
15. B.G. Buchanan and E.H. Shortliffe, *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley, 1984.
16. V.L. Yu, L.M. Fagan, S.M. Wraith, W.J. Clancey, A.C. Scott, J.F. Hanigan, R.L. Blum, B.G. Buchanan, and S.N. Cohen, 'Antimicrobial Selection by a Computer: a Blinded Evaluation by Infectious Disease Experts', *Journal of American Medical Association*, **242**, 1279–1282 (1979).
17. D.S.W. Tansley and C.C. Hayball, *Knowledge-Based Systems Analysis and Design*, Prentice-Hall, 1993.
18. M. Linster and M. Musen, 'Use of KADS to Create a Conceptual Model of the ONCOCIN task', *Knowledge Acquisition*, **4**, 55–88 (1992).
19. J.O. Coplien, 'Idioms and patterns as architectural literature', *IEEE Software*, 36–42 (1997).
20. C. Alexander, S. Ishikawa, S. Silverstein, I. Jacobsen, I. Fiksdahl-King, and S. Angel, *A Pattern Language*, Oxford University Press, 1977.
21. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
22. W. Clancey, 'Heuristic Classification', *Artificial Intelligence*, **27**, 289–350 (1985).
23. A.T. Schreiber, B.J. Wielinga, and J.M. Akkermans, 'Using KADS to Analyse Problem Solving Methods', in J. Balder and H. Akkermans (eds.), *Formal Methods for Knowledge Modeling in the CommonKADS Methodology: A Compilation (KADS-EE/T1.2/TR/ECN/014/1.0)*, Netherlands Energy Research Foundation, 1992, pp. 53–90.
24. V.R. Benjamins, 'Problem-solving methods for diagnosis and their role in knowledge acquisition', *International Journal of Expert Systems: Research & Applications*, **8**(2), 93–120 (1995).
25. R. Johnson, 'Documenting frameworks using patterns', *Proceedings of OOPSLA'92*, ACM SIGPLAN, 1992.

26. H. Akkermans, F. van Harmelen, G. Schreiber, and B. Wielinga, 'A Formalisation of Knowledge-Level Models for Knowledge Acquisition', in J. Balder and H. Akkermans (eds.), *Formal Methods for Knowledge Modeling in the CommonKADS Methodology: A Compilation (KADS-EE/T1.2/TR/ECN/014/1.0)*, Netherlands Energy Research Foundation, 1992, pp. 53–90.
27. C. Corbridge, N.P. Major, and N.R. Shadbolt, 'Models Exposed: An Empirical Study', *Proceedings of the 9th AAAI-Sponsored Banff Knowledge Acquisition for Knowledge Based Systems*, 1995.
28. M. Linster, 'A review of sisyphus 91 and 92: Models of problem-solving knowledge', in N. Aussenac, G. Boy, B. Gaines, M. Linser, J.-G. Ganascia, and Y. Kordratoff (eds.), *Knowledge Acquisition for Knowledge-Based Systems*, Springer-Verlag, 1992, pp. 159–182.
29. B. Bredeweg, 'Expertise in qualitative prediction of behaviour', *Ph.D. Thesis*, University of Amsterdam, 1992.
30. T.J. Menzies, 'Applications of abduction: Knowledge level modeling', *International Journal of Human Computer Studies*, **45**, 305–355 (September, 1996).
31. J. DeKleer and B.C. Williams, 'Diagnosing Multiple Faults', *Artificial Intelligence*, **32**, 97–130 (1987).
32. D. Poole, 'Probabilistic Horn abduction and Bayesian networks', *Artificial Intelligence*, **64**(1), 81–129 (1993).
33. W. Hamscher, L. Console, and J. DeKleer, *Readings in Model-Based Diagnosis*, Morgan Kaufmann, 1992.
34. T.J. Menzies and P. Compton, 'Applications of abduction: Hypothesis testing of neuroendocrinological qualitative compartmental models', *Artificial Intelligence in Medicine* (1997). To appear.
35. Z. Zdrahal and E. Motta, 'An In-Depth Analysis of Propose & Revise Problem Solving Methods', R. Mizoguchi, H. Motoda, J. Boose, B. Gaines, and P. Compton (eds.), *Proceedings of the Third Japanese Knowledge Acquisition for Knowledge-Based Systems Workshop: JKAW '94*, 1994.
36. E. Motta and Z. Zdrahal, 'The trouble with what: Issues in method-independent task specifications', *Proceedings of the 9th AAAI-Sponsored Banff Knowledge Acquisition for Knowledge-Based Systems Workshop Banff, Canada*, 1995.
37. R. Johnson. (personal communication).
38. T.C. Reeves, 'Evaluating what really matters in computer-based education', in M. Wild and D. Kirkpatrick (eds.), *Computer education: New Perspectives*, MASTEC, 1994, pp. 219–246.
39. J.L. Kolodner, 'Improving Human Decision Making Through Case-Based Decision Aiding', *AI Magazine*, 68 (1991).
40. A.J. Riel, *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.

Note that some of the Menzies references can be obtained from `http://www.cse.unsw.edu.au/~timm/pub/docs/papersonly.html`.