Submitted to the Handbook of Software Engineering and Knowledge Engineering,
Volume 2 (see http://www.ksi.edu/seke/hand.html)
URL= http://tim.menzies.com/pdf/00reuse.pdf.

Reuse in Software and Knowledge Engineering

Tim Menzies

Department of Electrical & Computer Engineering
University of British Columbia, Canada
http://tim.menzies.com; <tim@menzies.com>

December 14, 2000

1. Introduction

In the reuse approach to software construction, design is taken to be the re-shuffling of components developed previously, then abstracted into a reusable form. The idea of reusing old design work when doing new design work dates back at least to 1964 with Alexander's work on architecture [2, 3]. Contemporary expressions of this reuse approach include:

- Object-oriented design *patterns* [10, 42]
- Knowledge engineering research into *ontologies* [28, 31, 57];
- Knowledge engineering research into *problem solving methods* (PSMs) [12, 14, 44, 48].

Reusing patterns/ontologies/PSMs offers many advantages to designers. For example, suppose an analyst is reusing the financial knowledge shown in Figure 1. Supposing our analyst is reviewing a design for some point-of-sale system. Note that the background knowledge includes a "subsequent transaction" term. When browsing this knowledge, the analyst might be reminded to ask the question "are the sold items ever returned to the store?". That is, browsing reusable knowledge can assist in auditing and improving the current version of a system description.

Reuse of ontologies/patterns/PSMs comes at a cost. While reuse-based analysis can be exciting, it is an *economic issue* if reuse-based development is the best way to develop software. As we shall see, it is an open issue if the costs of reuse outweigh its benefits.

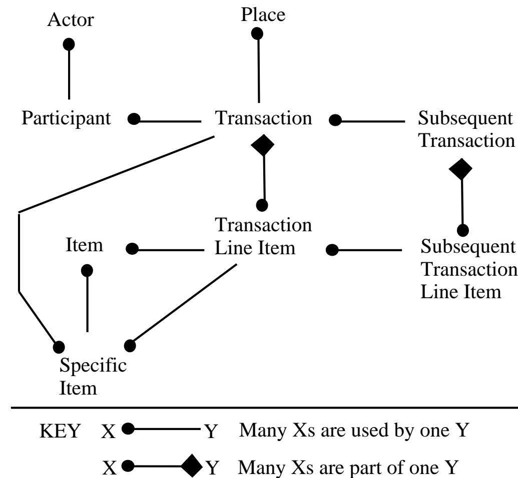

Figure 1: Part of Coad et.al.'s [16] definition of a financial transaction.

2. Reuse: the Benefits

There are many potential benefits of reuse. Some of these benefits include the *communications* benefit; the *interoperability* benefit; the browsing and searching benefit; the *systems engineering* benefit; and the *guidance* benefit.

2.1. *The Communications Benefit*

Any communication task is simplified by a shared lexicon. Such a lexicon can be developed by reusing and generalizing terms seen in prior applications. Generalized lexicons are called different things in different domains. Knowledge engineers typically call them *ontologies* while software engineers call them *patterns*.

Ontologies/patterns can be used to systematically view and share a specific topic/problem. For example, ontologies provide a unified framework within an organization that reduces the terminological confusion [56] arising from different contexts and viewpoints for a particular domain.

Attempting to reuse knowledge is a useful teaching tool. For example, reusing knowledge can simplify a student's task when (e.g.) reading textbooks. Also, studying reusable abstractions are useful when tutoring software or knowledge engineering [42]. Such abstractions serve as a useful final initiation ritual for a novice designer. When they "get" abstractions, we know that the students are capable of comparing and contrasting a wide range of systems.

2.2. *The Interoperability Benefit*

Interoperability among systems with different modeling methods, paradigms, languages and software tools can be achieved with ontologies that act as an inter-lingua [33].

2.3. *The Browsing/Searching Benefit*

The reusable knowledge within an ontology can assist an intelligent search engine with processing a query. For example, if a query returns no results, then the ontology could be used to automatically generalize the query to find nearest partial matches.

2.4. *The Systems Engineering Benefit*

If ontologies are the generalized *nouns* of a domain, problem solving methods (PSMs) are the generalized *verbs* from a domain. Such generalized verbs capture the Often the processing of a domain falls into cliched patterns of behavior. In knowledge engineering, these behavioral patterns are called PSMs.

Reusing ontologies and problem solving methods can simplify system development. Several examples are listed below from the knowledge engineering domain. More detailed examples from software engineering and knowledge engineering are offered later in this chapter:

- Kalfoglou executes the constraints found in existing ontologies to check new systems. Such pre-existing constraints, are a powerful tool for checking knowledge when other oracles are absent [30].
- One commercial company used the ontology associated with Motta's PSM design tools to formalize the regulations applicable to the design of the truck cabin [45]. This formalization, associated with a constraint analyzer, cut the design of the geometric layout of the cabin from 4 months to 1 day(!!).
- In the SPARK/ BURN/ FIREFIGHTER system, an intelligent PSM librarian was used to build nine KBS applications. Development times changed from one to 17 days (using the librarian) to a range of 63 to 250 days (without using the librarian) [38].
- The SALT editor used for the VT elevator configuration system restricted its knowledge editors to only those terms relevant for the propose-and-revise PSM used in VT [36, 37]. $^{2130}/_{3062} \approx 70\%$ of VT's rules could be auto-generated by SALT.
- RIME was an intelligent editor for simplifying the maintenance of DEC's XCON automatic computer configuration system. Bachant and McDermott [6] found that if a rule editor could access the ontology of the PSMs within XCON, then very large rules could be quickly built from very small specifications.

2.5. *The Guidance Benefit*

While we may use little of an ontology or a pattern or a PSM, it may still be useful as a "pointer tool". That is, the ontology/pattern/PSM could be used as a structuring tool for exploring a new domain. Roughly speaking, reusing abstracted forms of old knowledge is pointing the way saying "these kinds of things are important, even if these particular things are not". In this approach, developers kick-start the development with an ontology/pattern/PSM.

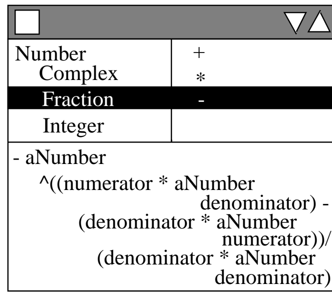

Figure 2: A class hierarchy browser

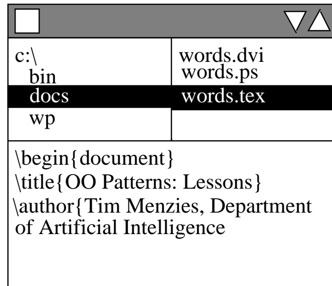

Figure 3: A disk browser

3. Examples of Reuse

To get a feel for the power of reuse-based design, the discussion now samples reuse work from the software engineering and knowledge engineering literature.

3.1. *Reuse in Software Engineering*

The work on object-oriented *patterns* is a good exemplar of reuse research in software engineering. Researchers in this area include the "gang-of-four"' (GOF) [26]; the "gang-of-five"(GOV) [10]; Fowler [23]; Shaw and Garlan [50]; and Coad et.al. [16].

Consider the class hierarchy browser of Figure 2. 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 3. 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.

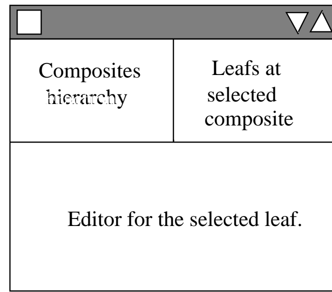

Figure 4: A composite browser

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 4). That is, our disk browser and class hierarchy browser are both presentations of nested composites.

Figure 5 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”.
- 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* [10]. 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 behavior*. In OO patterns: (i) the static components are described using class hierarchies and their relationships; and (ii) the runtime behaviors are described using some variant on collaboration diagrams [8]. The

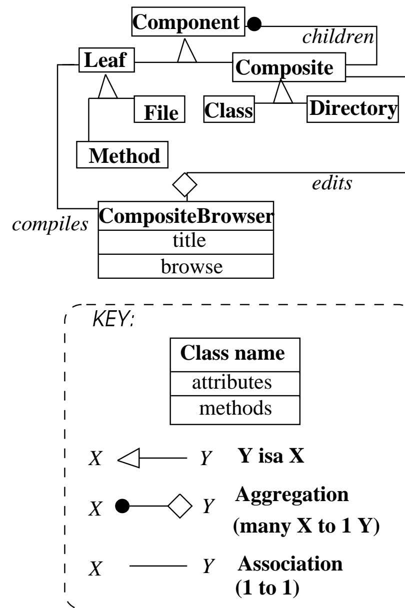

Figure 5: Object model for the composite browser

GOV argue that this format of a pattern is compatible with numerous other patterns researchers (page 11 of [10]).

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

1. Low-level language-dependent *idiom* patterns;

2. Middle-layer language independent *design* patterns describing a programmer's key mechanisms (e.g. the GOF patterns);

3. Top-level *architectural patterns* that spread across the entire application [50]. Examples of architectural patterns are *layered* architectures (e.g. the three tiered database-model-dialog 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 [23]) were also useful in a corporate finance applications (chapter 4 of [23]).

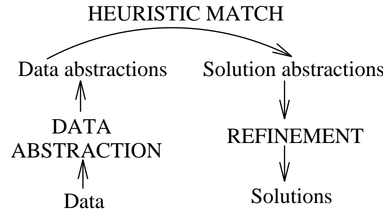

Figure 6: Heuristic classification

3.2. *Reuse in Knowledge Engineering*

OO patterns typically describe data structures that may repeat in many domains. Another kind of patterns are the behavioral patterns cataloged by knowledge engineers as problem solving methods (PSMs). The connection of patterns to PSMs is discussed elsewhere [42].

The goal of PSM modeling is to identify abstract reusable inference skeletons that appear in many expert systems; e.g. diagnosis, classification, monitoring, etc. Examples of this approach were listed above when discussing the systems engineering benefit of reuse. Other examples include generic tasks [13]; configurable role-limiting methods [27, 54]; model construction operators [15]; CommonKADS [48, 58]; the PROTEGE family of systems [22]; components of expertise [53]; MIKE [5]; and TINA [7].

The archetypal PSM is *heuristic classification*, first described by Clancey [14]. To find this PSM, 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 6).

For example, Figure 7 shows Clancey's analysis of the MYCIN [9, 60] 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 6.

Figure 8 shows Clancey's analysis of another system, written in LISP which diagnoses an electronic circuit in terms of the component that is causing faulty behavior. The abstract schema is shown on top, and an example is shown underneath. Note the similarity with Figure 7 and 6.

After the *Heuristic Classification* paper, Clancey refined his inference skeletons. In *Model Construction Operators* [15], Clancey argued that rules like Figure 9 contain domain-specific terminology (see Figure 10) as well as reusable inference strategies (see Figure 11). If these are removed from the rule, then not only have we isolated the true business knowledge in the rule (see Figure 12), 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 11; and (ii) a separate knowledge base containing the special domain heuristics like Figure 12.

Inspired by Clancey's work, subsequent researchers sought other abstract inference

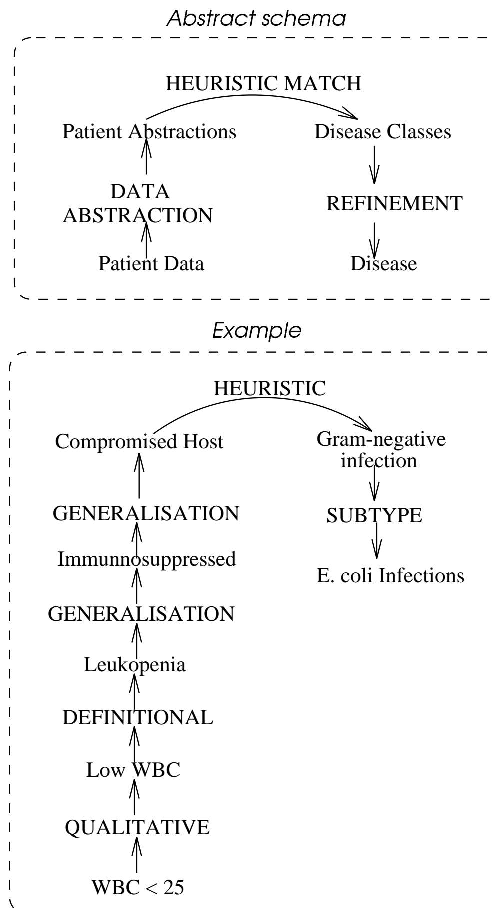

Figure 7: MYCIN

skeletons. Tansley & Hayball [55] list over two dozen reusable inference skeletons including systematic diagnosis (localization and causal tracing), mixed mode diagnosis, verification, correlation, assessment, monitoring, simple classification, heuristic classification, systematic refinement, prediction, prediction of behavior and values, design (hierarchical and incremental), configuration(simple and incremental) planning, and scheduling. These skeletons are recorded using the KADS notation of Figure 13 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 that Figures 6 or 13 do not imply a particular execution order of their func-

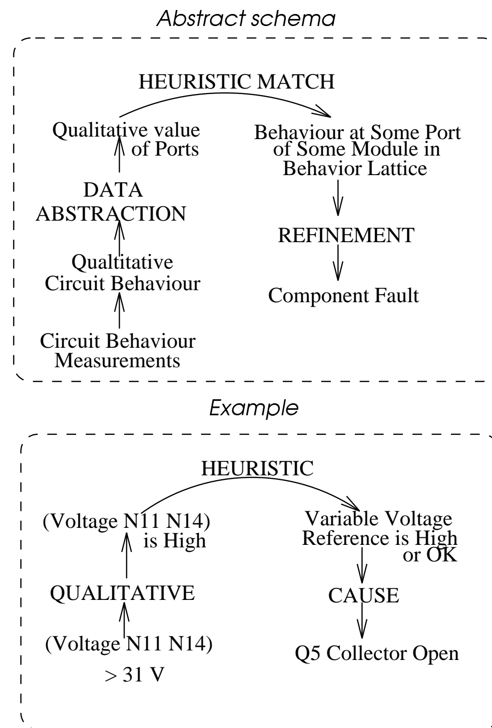

Figure 8: SOPHIE III

tions. Conceptually each function can be driven forwards or backwards to connect inputs to outputs or visa versa. The heuristic classification pattern of Figure 6 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. For more on KADS-style development, see [44].

```
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 9: A domain rule with hidden reusable inference fragments. From [15].

```
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 10: Domain-specific terms from Figure 9.

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 11: Problem solving strategies from Figure 9.

4. Problems with Reuse

Having discussed the *potential* uses of reuse, it is appropriate to next discuss the *reality* of reuse. Not all the benefits listed above have been achieved in practice. Even enthusiastic proponents of reuse note a puzzling lack of widespread reuse. For example:

> I do not think we have yet succeeded in software reuse. In the State of the Practice, I do not see a lot being applied. Yes, OO class frameworks, Java, Visual Basic, etc. have facilitated the code reuse, but less is done, in general, at the higher level such as in the design and requirements phases of a project. In the State of the Art, I have not perceived any incremental progress although many issues have been addressed [29].
>
> Reuse continues to be a problem whose potential remains elusive. Each new solution remains full of promise but riddled with what look like insurmountable problems. [46]

Even reuse enthusiasts such as Frakes [24] caution that there exist significant practical problems with the widespread proliferation of reuse libraries; e.g. problems with searching the reuse library.

A review of the literature supports the above claims. Frakes & Fox surveyed 100s of European and North American IT professionals to conclude that reuse levels were low (20% or less). Further, it was not correlated to technology (e.g. use of COBOL, C++, case tools, reuse libraries...) [25]. Reuse seemed to be correlated to non-software

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

Figure 12: The business knowledge of Figure 9.

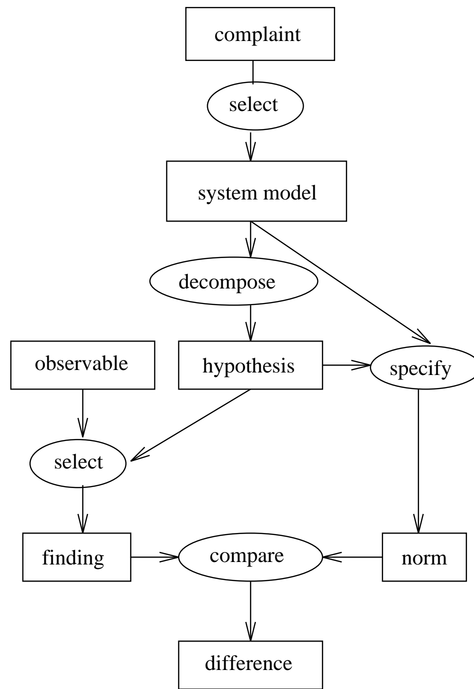

Figure 13: KADS: diagnosis

technology issues; e.g. hardware standards enabled high levels of reuse in the telecommunications industry (see Figure 14).

Studies in the knowledge engineering field suggest the same pattern of low-levels of reuse. Cohen et.al. documented the extent to which an ontology *supported* application development within DARPA's High Performance Knowledge Based Systems (HPKB) initiative [17]. *Support* was measured in terms of the words that appeared in some new application: if ⅔ of those words came from an ontology, then that ontology offered a 67% support for that application. Two teams were involved: one at SRI and one from Teknowledge who used the Cycorp knowledge base (hereafter CYC/Tek). The teams built applications using an upper ontology (UO) released by Cycorp. Along with the UO, CYC/Tek and SRI made their own local extensions. Both teams built and debugged their ontology using a set of sample questions (SQ) issued by the HPKB evaluation team. At a pre-announced date, 110 test questions (TQA) were issued and the applications were scored. After a brief respite, a scope change was announced, followed (several days later) by test questions for the new scope (TQC). The SRI system analyzed by Cohen *et.al* could only handle 40 of the 110 questions so the CYC/Tek results are divided into CYC/Tek(110) and CYC/Tek(40) where the latter is the subset of the CYC/Tek system relevant to the questions that SRI could handle. The results are shown in Figure 15. Note that the local ontological extensions supported new applications 3-4 times more than the UO terms; as the scope change (TQA-TQC) the

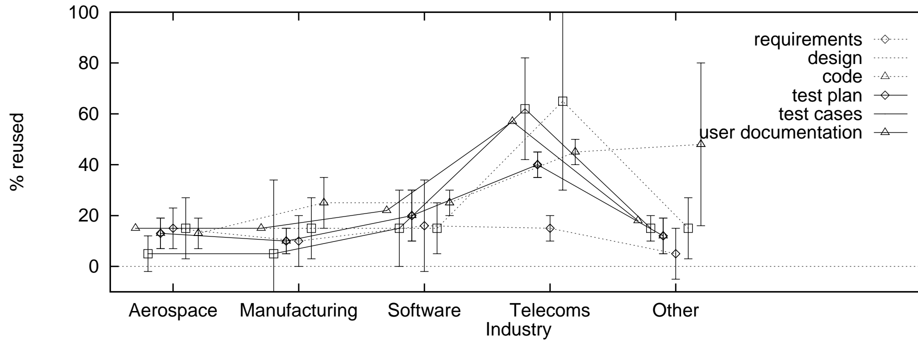

Figure 14: 95% confidence intervals for reuse levels in different industries.

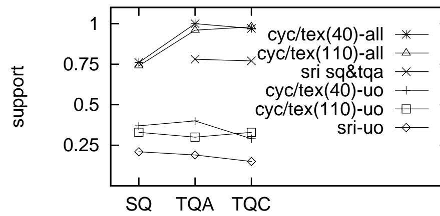

Figure 15: Reuse of ontologies

UO offered less and less support; and CYC/Tek's reuse of the UO was greater than that of SRI. These results suggested that the recent words you added yourself to an ontology offer more support than words added previously by other authors. That is, while developers might reuse their own work, they seem less likely to reuse the work of others.

So, despite the *potential* benefits of reuse, there are clearly problems with the *practice* of reuse. These problems include the *structuring* problem, the *productivity* problem, and the *stability* problem.

4.1. *The Structuring Problem*

Structuring the reusable knowledge is a open issue. Althoff argues that is hard to build reusable knowledge without considerable experience with the domain. The level of abstraction at which we formalize our reusable knowledge should be learned via extensive experience with that particular term [4]. Note that, according to Althoff, it is not necessarily true that reusable knowledge should always be expressed in some computer-readable form. Sometimes, simply rendering it on paper will suffice. For example, object-oriented "guidance patterns" serve to direct novice analysts to a set of issues that experienced analysts have found insightful. Such patterns include CHECKS [21], Caterpillar's Fate [32], and the strategies of Coad et.al. [16]. This type of reusable knowledge is stored as simple checklists of English text.

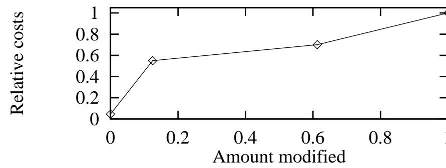

Figure 16: COCOMO-II, the cost of reuse with X% changes

4.2. *The Productivity Problem*

Another major problem is that reuse is not necessarily a more productive method of building systems. For example, the COCOMO-II software cost estimation model offers an estimate of the cost of adapting reusable sub-routines for a new project [1, p21]. That model argues that a learning curve must be traversed before a module can be adapted. By the time you know enough to change a little of that module, you may as well have re-written 60% of it from scratch; see Figure 16.

As another example, consider the Corbridge et.al. [20] study. In that study, international KA experts used PSMs to guide their analysis of a transcript of a patient talking to a doctor [20]. One group used a diagnosis PSM matured over many years; another used an abstract model invented very quickly (the "straw man"); and the rest used no model at all. The results are shown in Figure 17. The "mature model" group performed as well as the "straw man" group. Further, the "no model" group out-performed the groups using the reuse models.

Reuse Model	% disorders identified	% knowledge fragments identified
Straw man: invented very quickly	50	28
Mature model: decades of work	55	34
No model	75	41

Figure 17: Productivity using different models.

Further, note that successful reuse can actually *decrease* overall productivity. Consider a reuse library containing a bug. That error would be injected into every application that uses that library. For example, in the Sisyphus-II experiments, various research groups re-implemented part of the VT elevator configuration system [36]. All the groups implemented a PSM with the same error [61]. The Sisyphus-II propose-and-revise PSM was a local greedy search. Local hill-climbing may ignore solutions

	ft/min				
Capacity (lbs)	200	250	300	350	400
2000	√	√	×	√	√
2500	×	×	√	√	√
3000	×	√	√	√	√
3500	√	×	×	×	×
4000	×	×	×	×	×

Figure 18: PSMs succeeding (√) or failing (×) to configure an elevator.

which are initially unpromising, but lead later on to better solutions. In one experiment, this local greedy search algorithm failed to configure $^{13}/_{25}$ elevator configurations: see Figure 18. Only one of the Sisyphus-II groups reported this error. Apparently, the rest trusted their reusable PSM so much, that they did not perform detailed validation studies. If other developers are as complacent about their reuse libraries, then reuse could decrease overall software productivity since they will spend more time chasing bugs introduced via reuse.

When discussing above the *Systems Engineering Benefit*, various studies were presented that seem to refute the thesis of this section. If reuse complicates productivity measures so much, why is it that Motta et.al. can say that example of truck cabin design, the time to develop the layout dropped from 4 months to 1 day? It is hard to assess such anecdotal evidence without more precise metrics collection [39,40]. For example, when Motta's colleagues reduced their design time from 4 months to 1 day, how much of that reduction was due to the PSM framework and how much to the constraint analyzer used in that domain? Elsewhere [43], I have criticized overly-enthusiastic reports of reuse that lack detailed measures*. For example, proponents of reuse rarely track the on-going costs of maintaining with those components [41] (exception: [34]). Most reuse reports do not clearly distinguish between *verbatim reuse* and *reuse with some tinkering*. Recalling Figure 16, such tinkering to customize a reusable component can significantly increase the cost and decrease the benefits of using reusable components.

4.3. *The Stability Problem*

If a reuse library is *unstable*, then it will be continually rewritten. The cost of extensive rewrites can negate the economic benefits of reuse.

Knowledge is often unstable and this instability can produce dramatic changes to knowledge. For example, half of XCON's thousands of rules were changed every year [52]. To some extent, this might be due to its changing operational requirements (XCON configured computers for DEC and DEC keeps releasing new computers). However, even in supposedly stable domains, knowledge keeps being patched. Garvin ES-1 [18] offered interpretations of biochemical results. Over its lifetime, the biochemical assay hardware remained constant and, presumably, humans did not evolve significantly. Yet KB maintenance was on-going. The kind of (dramatic) changes seen within that KB are shown in Figure 19. The change in KB size of Gavrin ES-1 is shown in Figure 20. Note that the rate of change within this system was linear; i.e. even in a stable domain, knowledge kept changing†.

What could cause instability in knowledge? One explanation is that consensus expert knowledge is hard to find. Hence, any attempt to record such a consensus implies a constant "pursuit and patch" of feuding ideas. There is some evidence that experts disagree, even with themselves. Shaw used a terminology checking tool called repertory grids to compare the meaning of terms used by three geology experts on a common problem [51]. Two experiments were performed. In the calibrating experiment, experts

*One of the few reuse reports that includes quality measures is [34]. However, that report refer to intra-institutional reuse, not widespread inter-institutional reuse.

†Technically, the Garvin ES-1 size changes are also consistent with a logarithmic curve. However, a visual inspection of the plot strongly suggests a linear fit.

A. Originally

```
 RULE(22310.01) IF  (bhthy or
     utsh_bhft4 or
     vhthy) and not on_t4
           and not surgery
           and (antithyroid or
                hyperthyroid)
THEN DIAGNOSIS("...thyrotoxicosis")
```

B. Same rule, 3 years later

```
 RULE(22310.01) IF  ((((T3 is missing)
       or (T3 is low and
           T3_BORD is low))
       and TSH is missing
       and vhthy
       and not (query_t4 or on_t4 or
                 or surgery or tumour
                 or antithyroid
                 or hypothyroid
                 or hyperthyroid))
      or ((((utsh_bhft4 or
             (Hythe and T4 is missing
              and TSH is missing))
             and (antithyroid or
                  hyperthyroid))
          or  utsh_bhft4
          or  ((Hythe or borthy)
                and T3 is missing
                and (TSH is undetect
                     or TSH is low)))
          and not on_t4 and not
              (tumour or surgery)))
     and (TT4 isnt low or T4U isnt low)
THEN DIAGNOSIS("...thyrotoxicosis")
```

Figure 19: A rule maintained for 3 years.

reviewed their own knowledge, 12 weeks after they created it. This first experiment gives baseline expected agreement figures for a repertory grid analysis (see Figure 21). In the second experiment, inter-expert agreement was analyzed (see Figure 22). Note (e.g.) E_1,E_3: the results were much lower than in the calibration experiment suggesting that these experts held very different views about a supposedly standard problem in their field.

Another reason for instability of knowledge is that new experience always gives new insights which significantly change old knowledge. This may be true for both human experts and automatic machine learners:

- Shalin et.al. [49] tried to find "accepted practice"; i.e. reused knowledge within expert communities. They found that experts do modify their behavior according to community standards of "accepted practice". However, it is only novices who slavishly re-apply that accepted practice. Experts adapt accepted practice when they apply it. That is, experts:
  - Partially match current problem to libraries of accepted practice.
  - Implement an acceptance test for their adaptation.
  - Modify the accepted practice library if acceptance failure.
- Catlett [11] used C4.5 [47] to learn decision trees for 11 problems using either all the N=3000..5000 training cases or half the cases (randomly selected). The change in tree size and error rates are shown in Figure 23. In all but 1 case

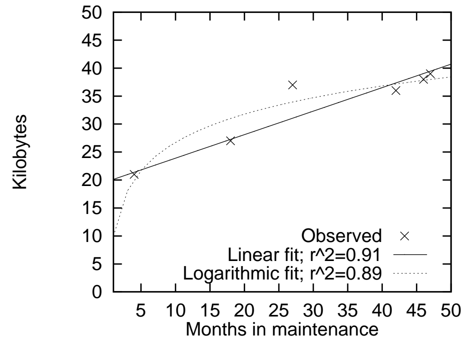

Figure 20: KB changes in Garvin ES-1.

Expert	%understands	%agrees
E_1	62.5	81.2
E_2	77.8	94.4
E_3	85.7	78.6

Figure 21: Self-agreement, 12 weeks later

Expertpairs	%understands	%agrees
E_1, E_2	62.5	33.3
E_2,E_1	61.1	26.7
E_1,E_3	31.2	8.3
E_3,E_1	42.9	33.3
E_2,E_3	44.4	20.0
E_3,E_2	71.4	33.3

Figure 22: Inter-expert agreement

domain	Change in tree size	Change in classification error of the learned tree
demon	0.97	0.51
wave	1.91	0.95
diff	1.46	0.69
othello	1.68	0.8
heart	1.61	0.65
sleep	1.73	0.91
hyper	1.74	0.83
hypo	1.45	0.85
binding	1.51	0.82
replace	1.38	0.8
euthy	1.33	0.61
mean	1.52	0.77

Figure 23: Impact of learning a decision tree from N or $2N$ examples.

(demon, first row), more experience meant significantly less errors, but larger theories.

The problem of knowledge instability has been seen in PSM libraries. Elsewhere [42], I have analyzed eight different supposedly reusable models of diagnosis (four from the PSM community, four from elsewhere). While some of these views on diagnosis share some common features, they reflect fundamentally divergent views on how to perform diagnosis. I therefore believe that, at least in the case of diagnosis, the consensus view has yet to stabilized and may not do so in the near future. More generally, I'm not sure that a consensus view on any of the PSMs has been reached, despite decades of research. There are significant differences between the list of PSM primitives offered by Clancey [15], KADS [59], and SBF. Also, the number and nature of the inference knowledge is not fixed. Often when a domain is analysed, a new PSM is induced [35].

5. Discussion

The quest for appropriate reusable knowledge is fundamental to the western scientific tradition. Compton traces this quest back to the ancient Greeks:

> The reductionist assumption that one should be able to dig deep enough to find primitive concepts and the relationships between them on which knowledge is built finds its origins in Plato's concept of archetypes. That is, that there exist (literally) archetypes for all the things in the world and the concepts we use. Proposals such as (reuse) are essentially statements of belief that if the archetypes and relevant logical relationships can be found and manipulated intelligent thought can be reproduced [19, p280].

This chapter has offered numerous examples of reuse, and discussed the potential benefits and associated costs of reuse. In summary, reuse offers designers and educates a communications benefit, an interoperability benefit, and a browsing and search benefit. Reuse can *potentially* offer a systems engineering benefit and a guidance benefit. However, the review of the literature presented here suggests that it is at least an open issue if these last two potential benefits have been realized.

References

1. C. Abts, B. Clark, S. Devnani-Chulani, E. Horowitz, R. Madachy, D. Reifer, R. Selby, and B. Steece. COCOMO II Model Definition Manual. Technical report, Center for Software Engineering, USC,, 1998. `http://sunset.usc.edu/COCOMOII/cocomox.html#downloads`.
2. C. Alexander. *Notes on Synthesis of Form*. Harvard University Press, 1964.
3. C. Alexander, S. Ishikawa, S. Silverstein, I. Jacobsen, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, 1977.
4. K.-D. Althoff, M. Nick, and C. Tautz. Improving Organizational Memories Through User Feedback. In F. Bomarius, editor, *Proc. of the Workshop on Learning Software Organizations (LSO) (in conjunction with the11th International Conference on Software Engineering and Knowledge Engineering, SEKE'99, Kaiserslauten, Germany)*, pages 27–44, June 1999.
5. J. Angele, D. Fensel, and R. Studer. Domain and Task Modelling in MIKE. In A. Sutcliffe et.al., editor, *Domain Knowledge for Interactive System Design*. Chapman & Hall, 1996.

6. J. Bachant and J. McDermott. R1 Revisited: Four Years in the Trenches. *AI Magazine*, pages 21–32, Fall 1984.
7. 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.
8. 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`.
9. B.G. Buchanan and E.H. Shortliffe. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, 1984.
10. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.
11. J. Catlett. Inductive learning from subsets or Disposal of excess training data considered harmful. In *Australian Workshop on Knowledge Acqusition for Knowledge-Based Systems, Pokolbin*, pages 53–67, 1991.
12. B. Chandrasekaran. Towards a Taxonomy of Problem Solving Types. *AI Magazine*, pages 9–17, Winter/Spring 1983.
13. B. Chandrasekaran, T.R. Johnson, and J. W. Smith. Task Structure Analysis for Knowledge Modeling. *Communications of the ACM*, 35(9):124–137, 1992.
14. W. Clancey. Heuristic Classification. *Artificial Intelligence*, 27:289–350, 1985.
15. W.J. Clancey. Model Construction Operators. *Artificial Intelligence*, 53:1–115, 1992.
16. P. Coad, D. North, and M. Mayfield. *Object Models: Strategies, Patterns, and Applications*. Prentice Hall, 1997.
17. P. Cohen, V. Chaudhri, A. Pease, and R. Schrag. Does Prior Knowledge Facilitate the Development of Knowledge-based Systems? In *AAAI'99*, 1999.
18. P. Compton, K. Horn, J.R. Quinlan, and L. Lazarus. Maintaining an Expert System. In J.R. Quinlan, editor, *Applications of Expert Systems*, pages 366–385. Addison Wesley, 1989.
19. P. Compton, B. Kang, P. Preston, and M. Mulholland. Knowledge Acquisition Without Analysis. In *European Knowledge Acquisition Workshop*, 1993.
20. C. Corbridge, N.P. Major, and N.R. Shadbolt. Models Exposed: An Empirical Study. In *Proceedings of the 9th AAAI-Sponsored Banff Knowledge Acquisition for Knowledge Based Systems*, 1995.
21. W. Cunningham. The CHECKS Pattern Language of Information Integrity. In J. Coplien and D. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, 1995. Also avialable at *http://c2.com/ppr/checks.html*.
22. H. Eriksson, Y. Shahar, S. W. Tu, A. R. Puerta, and M. A. Musen. Task Modeling with Reusable Problem-Solving Methods. *Artificial Intelligence*, 79(2):293–326, 1995.
23. M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison Wesley, 1997.
24. W. Frakes. Domain engineering education. In *Proceedings of WISR9: The 9th annual workshop on Institutionalizing Software Reuse*, 1999. Available from `http://www/umcs.maine.edu/~ftp/wisr/wisr9/final-papers/Frakes.html`.
25. W.B. Frakes and C.J. Fox. Sixteen Questions About Software Reuse. *Communications of the ACM*, 38(6):75–87, June 1995.
26. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
27. Y. Gil and E. Melz. Explicit Representations of Problem-Soving Strategies to

Support Knowledge Acquisition. In *Proceedings AAAI' 96*, 1996.

28. T.R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.
29. E. Guerrieri. Reuse success - when and how? In *Proceedings of WISR9: The 9th annual workshop on Institutionalizing Software Reuse*, 1999. Available from `http://www/umcs.maine.edu/~ftp/wisr/wisr9/final-papers/Guerrieri.html`.
30. Kalfoglou,Y. and Robertson,D. A Case Study in Applying Ontologies to Augment and Reason about the Correctness of Specifications. In *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering, SEKE'99, Kaiserslauten, Germany*, pages 64–71, June 1999.
31. Y. Kalgoglou. Ontologies in Software Design. In S.K. Chung, editor, *Handbook of Software and Knowledge Engineering (volume 1)*, 2001.
32. N. Kerth. Caterpillar's Fate: A Pattern Language for Transformation from Analysis to Design. In J. Coplien and D. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, 1995. Also available from *http://c2.com/ppr/catsfate.html*.
33. J. Lee, M. Gruninger, Y. Jin, T. Malone, A. Tate, G Yost, and other members of the PIF working group. The PIF Process Interchange Format and framework. *Knowledge Engineering Review*, 13(1):91–120, February 1998.
34. W. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, pages 23–30, 1994.
35. M. Linster and M. Musen. Use of KADS to Create a Conceptual Model of the ONCOCIN task. *Knowledge Acquisition*, 4:55–88, 1 1992.
36. S. Marcus and J. McDermott. SALT: A Knowledge Acquisition Language for Propose-and-Revise Systems. *Artificial Intelligence*, 39:1–37, 1 1989.
37. S. Marcus, J. Stout, and J. McDermott. VT: An Expert Elevator Designer That Uses Knowledge-Based Backtracking. *AI Magazine*, pages 41–58, Winter 1987.
38. D. Marques, G. Dallemagne, G. Kliner, J. McDermott, and D. Tung. Easy Programming: Empowering People to Build Their Own Applications. *IEEE Expert*, pages 16–29, June 1992.
39. T. Menzies. hQkb- The High Quality Knowledge Base Initiative (Sisyphus V: Learning Design Assessment Knowledge). In *KAW'99: the 12th Workshop on Knowledge Acquisition, Modeling and Management, Voyager Inn, Banff, Alberta, Canada Oct 16-22, 1999*, 1999. Available from `http://tim.menzies.com/pdf/99hqkb.pdf`.
40. T. Menzies, K.D. Althoff, Y. Kalfoglou, and E. Motta. Issues with Meta-Knowledge. *International Journal of Software Engineering and Knowledge Engineering*, 10(4), August 2000. Available from `http://tim.menzies.com/pdf/00sekej.pdf`.
41. Tim Menzies, Bojan Cukic, Harhsinder Singh, and John Powell. Testing Nondeterminate Systems. In *ISSRE 2000*, 2000. Available from `http://tim.menzies.com/pdf/00issre.pdf`.
42. T.J. Menzies. OO Patterns: Lessons from Expert Systems. *Software Practice & Experience*, 27(12):1457–1478, December 1997. Available from `http://tim.menzies.com/pdf/97patern.pdf`.
43. T.J. Menzies. Towards Situated Knowledge Acquisition. *International Journal of Human-Computer Studies*, 49:867–893, 1998. Available from `http://tim.menzies.com/pdf/98ijhcs.pdf`.
44. E. Motta. The Knowledge Modelling Paradigm in Knowledge Engineering. In

S.K. Chung, editor, *Handbook of Software and Knowledge Engineering (volume 1)*, 2001.
45. E. Motta and Z. Zdrahal. A library of problem-solving components based on the intergration of the search paradigm with task and method ontologies. *International Journal of Human Computer Studies*, 49:437–470, 1998.
46. D. Perry. Some holes in the emperor's reused clothes. In *Proceedings of WISR9: The 9th annual workshop on Institutionalizing Software Reuse*, 1999. Available from `http://www/umcs.maine.edu/~ftp/wisr/wisr9/final-papers/Perry.html`.
47. R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1:81–106, 1986.
48. A. TH. Schreiber, B. Wielinga, J. M. Akkermans, W. Van De Velde, and R. de Hoog. CommonKADS. A Comprehensive Methodology for KBS Development. *IEEE Expert*, 9(6):28–37, 1994.
49. V.L. Shalin, N.D. Geddes, D. Bertram, M.A. Szczepkowski, and D. Dubois. Expertise in Dynamic, Physical Task Domains. In P.J. Feltovich, K.M. Ford, and R.R. Hoffman, editors, *Expertise in Context*, chapter 9, pages 195–217. MIT PRess, 1997.
50. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
51. M.L.G. Shaw. Validation in a Knowledge Acquisition System with Multiple Experts. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1259–1266, 1988.
52. E. Soloway, J. Bachant, and K. Jensen. Assessing the Maintainability of XCON-in-RIME: Coping with the Problems of a VERY Large Rule-Base. In *AAAI '87*, pages 824–829, 1987.
53. L. Steels. Components of Expertise. *AI Magazine*, 11:29–49, 2 1990.
54. B. Swartout and Y. Gill. Flexible Knowledge Acquisition Through Explicit Representation of Knowledge Roles. In *1996 AAAI Spring Symposium on Acquisition, Learning, and Demonstration: Automating Tasks for Users*, 1996.
55. D.S.W. Tansley and C.C. Hayball. *Knowledge-Based Systems Analysis and Design*. Prentice-Hall, 1993.
56. M. Uschold, M. King, S. Moralee, and Y. Zorgios. The enterprise ontology. *The Knowledge Engineering Review*, 13(1), February 1998.
57. G. van Heust, A. Th. Schreiber, and B.J. Wielinga. Using explicit ontologies in KBS development. *International Journal of Human Computer Studies*, 45:183–292, 1997.
58. B.J. Wielinga, A.T. Schreiber, and J.A. Breuker. KADS: a Modeling Approach to Knowledge Engineering. *Knowledge Acquisition*, 4:1–162, 1 1992.
59. B.J. Wielinga, A.T. Schreiber, and J.A. Breuker. KADS: a Modeling Approach to Knowledge Engineering. *Knowledge Acquisition*, 4:1–162, 1 1992.
60. 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.
61. Z. Zdrahal and E. Motta. Improving Conpetence by Intergrating Case-Based Reasoning and Heuristic Search. In *10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop, November 9-14, 1996, Banff, Canada*, 1996.