Comparing and Generalising Models for Metrics Repositories

Tim Menzies * Sita Ramakrishnan †

October 13, 1996

Abstract

Methodology assessment requires metrics. Large scale metrics collection requires a persistent data store. Persistent data stores require a model for the database design. Such models reflect the assumptions of their designers; e.g. waterfall development, non-object-oriented languages, etc. We review several models for metrics repositories to conclude that most of the existing metrics repositories are not suitable for software developed using an iterative, object-oriented, use-case driven approach. Based on this analysis, we have evolved RA-2: our preferred repository model. RA-2 is designed to be useful for both object-oriented and non-object-oriented systems.

*Department of Artificial Intelligence, School of Computer Science and Engineering, University of New South Wales, PO Box 1, Kensington, Sydney, Australia, 2033 timm@cse.unsw.edu.au http://www.cse.unsw.edu.au/~timm

†Department of Software Development, Monash University, Caulfield, Melbourne, Australia sitar@insect.monash.edu.au http://www.sd.monash.edu.au/~sitar

1 Introduction

There is a pressing current need for empirical investigations of software engineering in general [5] and object-oriented (OO) systems in particular [15, 17]. If these empirical investigations are concerned with software lifecycle issues, then these investigations require some storage device for metrics collected during the entire lifecycle. That is, empirical investigations of software engineering require a model for a persistent data store to track software over its entire life.

There is very little in the literature concerning the design of such data stores. Of the models we could find [3, 8, 12, 13, 18, 21], all had a different emphasis. For example, some of them focused on portions of the Fenton [6, chpt. 3] metrics triad:

- The RA-1 model [18] is *process*-centric; i.e. it monitors the manner in which software is constructed;
- The TAME resource model [12] is, as its name suggests, focuses on the *resources* used to generate the product;
- The Harrison model [8] is *product*-centric and only monitors details of the constructed entity.

Further, aside from the one we designed (the RA-1 model [18]), none of these models were suitable for iterative, use-case driven, OO development.

Based on our analysis of these models, we have developed the RA-2 repository model (§2). RA-2 is suitable for OO systems. It includes a use case model (§2.1) and offers support for software iteration (§3). Unlike the other models reviewed here (§4), RA-2 supports metric collection for resource (§2.2) metrics *and* process (§2.3) metrics

and product (§2.4). Also, in order to assess the developed software, it includes a bug and enhancement tracking system (§2.5).

RA-2 makes minimal assumptions about the structure of the product being developed. Hence, we believe that it is useful for general software metrics collection and not just OO (§3).

Notation conventions: The RA-2 model will be shown using the notation of Rational's Unified Notion version 0.8. Text shown in this SMALL CAPS font will become entities or relationships in RA-2. Text shown in this `typewriter` font discusses data entered into the RA-2 model. Text shown in this san serif font are entities from other model repositories than RA-2. When the rules of grammar demand it, we take some liberties with entity/class names. For example, we may write ACTIVITIES when, strictly speaking, we should write ACTIVITYS.

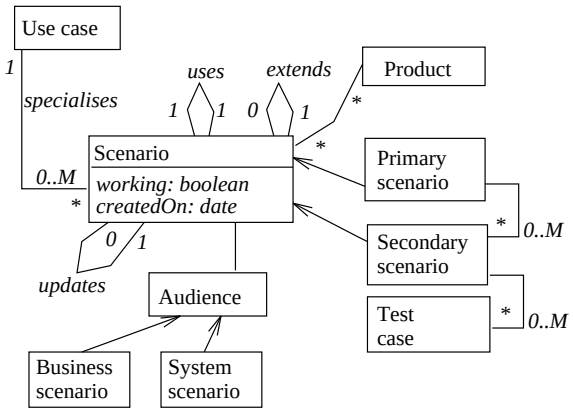

Figure 1: RA-2: USE CASES.

2 RA-2

2.1 Use-Cases

OO development is typically iterative. There are two reasons for this:

- Parnas [16] cautions that for many real real systems, in order to get the *whole* picture, the design has to leave some *holes* to be filled in at a later stage. This observation is central to the OO approach. The information hiding properties of class encapsulation permit extensive internal reorganisations of a design while maintaining a working system.

- The conceptual model used for analysis (classes and methods) is the same conceptual model used in design and implementation. Hence, the gap between analysis and implementation is much less than between (e.g.) DFDs to a COBOL program. Experience gained during the implementation can be quickly converted into design revisions. Consequently, OO designs can be quickly revised.

A common technique for handling iterative development and user-specifications in OO systems is the use case [2, 11, 19, 20]. Use cases have become the main driver of OO development [10]. They are used as the basis of specifying the functional requirements, defining software objects, allocating functions to objects, and designing the interface [10]. For a short tutorial introduction to use cases, see [20]. For an intricate use of use cases for tracing the link between requirements and code, see [19].

Our proposed model of use cases is based on Booch [2] (see Figure 1). A USE CASE is a general description of the events surrounding a set of actors (e.g. booking a flight). When the details of that use case are filled in, we generate many SCENARIOS (e.g. Tim books a national flight to Sydney; Sita books an international flight to Delhi). SCENARIOS have different AUDIENCES. BUSINESS SCENARIOS are comprehensible to a business user while a SYSTEM SCENARIO refers to some "under the hood" processing which we will shield from the business user. A SCENARIO is either a PRIMARY SCENARIO (where no exception conditions arise), or a SECONDARY SCENARIO (where the typical exceptions conditions arise). If all exceptions conditions are fully enumerated, then the SECONDARY SCENARIO becomes a TEST CASE. SCENARIOS can be defined in terms of other scenarios using the *uses* relationship (which always uses another scenario) or an *extends* relationship in which one scenario may or may not branch to another scenario.

USE CASES can be used at all stages of the software development process; e.g. during elicitation to capture user requirements; during phys-

ical design to audit class designs; during validation to assess the working system; and during auditing when tracing code fragments to user requirements [19]. Further, a graph of % SCENARIOS covered by the current system versus time is a powerful tool for monitoring an iterative development process.

Since they are the primary control tool for OO projects, we have to carefully monitor SCENARIO creation, SCENARIO modification, and the business user signing off that the SCENARIO is adequately WORKING). RA-2 takes the following Draconian approach: after a SCENARIO is added to the repository, it can't be modified. If a modification is required, then a new scenario is created and an UPDATES link is added from the new SCENARIO back to the old SCENARIO. At any point in time, the *current* SCENARIOS are those with no UPDATE link.

The relationship between CLASSES and SCENARIOS is intricate. A single SCENARIO may require numerous CLASSES to implement it and the same CLASS may be be used in many scenarios. Also, while use cases have been developed in the OO field, we believe that they are a general tool for any prototyping approach. Hence, for the purposes of generality, we relate SCENARIO to the more general term PRODUCT (in RA-2, CLASS is a subclass of PRODUCT Figure 5).

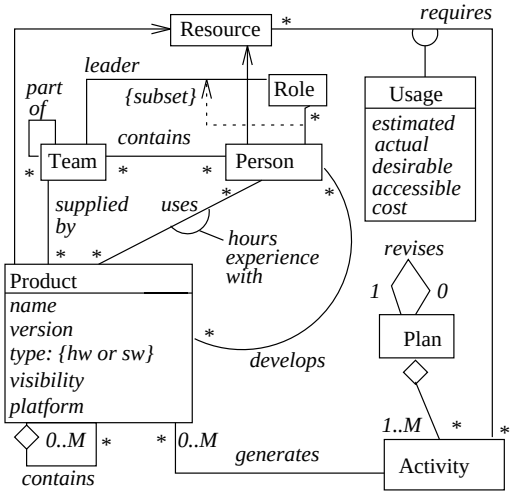

Figure 2: RA-2: RESOURCES

2.2 Resources

Three major resources in any system development are PERSONS, PRODUCTS, and time (see Figure 2). PERSONS work in TEAMS. TEAMS can be part of larger teams and so-on recursively (e.g. the `interface team` of the `smelter project team` of the `Information Technology Division team` of the `IBM Australia team` of the `IBM International team`). PERSONS can have many ROLES, one of which is team LEADER.

A PERSON USES existing PRODUCTS to DEVELOP new PRODUCTS. This PERSON has so many hours of experience with this PRODUCT. PRODUCTS have a NAME, a VERSION number, and a PLATFORM they run on. PRODUCTS may be software or hardware[1]. PRODUCTS may be aggregations of other PRODUCTS. PRODUCTS come from a supplier, which are other TEAMS (e.g. the `Microsoft` team produces the PRODUCT `Microsoft Office` which, in turn, contains other PRODUCTS such as `Word, Powerpoint, Access`, etc.). In terms of looking inside/ controlling a PRODUCT, a PRODUCT has four levels of VISIBILITY in increasing order:

1. None. For example, PERSONS may have no VISIBILITY of legacy or COTS software (commercial, off the shelf systems).
2. Customisation via parameter setting.
3. Customisation via scripting language; e.g. `Microsoft Word` comes with the scripting language `Word Basic`.
4. Full source code; i.e. full VISIBILITY.

PERSONS develop PRODUCTS via a PLAN. PLANS change and so they may be REVISED by a subsequent plan. In RA-2 time is modeled in the time attributes in the USAGE record.

ACTIVITIES need RESOURCES. The ESTIMATED time for USING a RESOURCE may be different to the ACTUAL time spent with it due to (i) the complexities of software estimation; (ii) the COST of

[1] There is an important class of tools used by software developers that are not hardware or software. A information processing system may contain many components, only some of which are hardware or software. For example, a software methodology is a tool, even though it may not be embodied in hardware or software. RA-2 models such methodological tools as ACTIVITY networks. See Figure 3

the RESOURCE; (iii) other limits to the hours of its ACCESSIBILITY. The ACTUAL USAGE time could be filled in from a timesheet system (not modeled in RA-2).

2.3 Process

A process is realised by a set of ACTIVITIES (see Figure 3). Software processes are characterised by the network of NAMED ACTIVITIES they propose:

- ACTIVITIES are recursive; i.e. ACTIVITIES have SUB-ACTIVITIES, which themselves are ACTIVITIES so they can have sub-sub-ACTIVITIES and so on. The ESTIMATED-HOURS of a super-ACTIVITY is the sum of the ESTIMATEDHOURS of its SUB-ACTIVITIES.
- An ACTIVITY may REQUIRE (i) other ACTIVITIES to be completed, or (ii) other RESOURCES to be available before this one can start.

The STARTDATE and STOPDATES of an ACTIVITY can be set by the user or deduced from a recursive traversal of the SUB-ACTIVITIES and REQUIRES link.

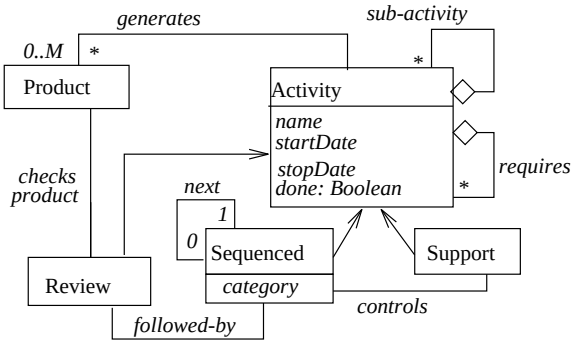

Figure 3: RA-2: ACTIVITIES

Processes contain combinations of SEQUENCED, SUPPORT, or REVIEW ACTIVITIES:

- SEQUENCED ACTIVITIES occur in some order according to the methodology being used. A SEQUENCED activity is on the PLAN (see Figure 2) and has a checkpoint where a REVIEW of its PRODUCTS is performed.
- REVIEW task lets us model checkpoints in the process where the project may be halted if progress is not satisfactory. Such *commit partition* points are essential part of risk-driven, iterative spiral software process [1].
- SUPPORT ACTIVITIES tasks are those which occur in parallel with any project such as project monitoring & control, quality management, document development, training, and configuration management (including version control and backups). These are shown as the WATCH! category in Figure 4.

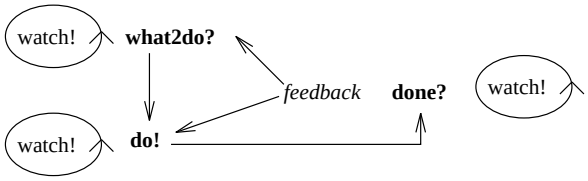

Figure 4: A network of instances of ACTIVITIES. SEQUENCED CATEGORIES are one of WHAT2DO?, DO!, DONE?, WATCH!.

After Hodgson [9], we categorise SEQUENCED ACTIVITIES into the three meta-groups shown in Figure 4:

- In the WHAT2DO? logical design ACTIVITY CATEGORY, a brainstorming/elicitation sub-activity is followed by an ordering/representation sub-activity.
- In the DO! physical design ACTIVITY CATEGORY, the idealised logical design is typically compromised as it is contorted to fit into some lower-level representation.
- In the DONE? test ACTIVITY CATEGORY, the results of the physical design are assessed. Testing can be divided into *verification* (i.e. was the system built right?) and *validation* (i.e. was the right system built?). Feedback from the testing process can improve the physical and logical designs.

Hodgson argues that the CATEGORIES of Figure 4 are recursive; i.e. the work involved within any of the SEQUENCED ACTIVITIES can be sub-divided

up into WHAT2DO?, DO!, DONE?. For example, in the verification SUB-ACTIVITY, the tests have to be planned (WHAT2DO?), implemented in some programming language (DO!), then compiled, run and their results evaluated (DONE?).

2.4 Products

The RA-2 PRODUCTS model is shown in Figure 5. PRODUCTS are stored in FILES by their AUTHORS. One issue in iterative design is maintaining traceability between CODE artefacts and the requirements DOCUMENTS that generated them [19]. Hence, the connection between DOCUMENT PRODUCTS and the CODE PRODUCTS they discuss must be maintained. Not all DOCUMENTS are directly related to CODE. For example, the release plan or the business case justifying the development may have required *discusses* link to a CODE PRODUCT.

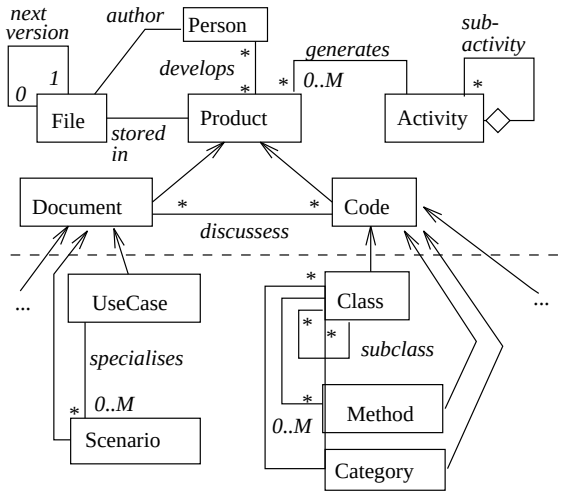

Figure 5: RA-2: PRODUCTS. The dotted line is explained elsewhere (§3).

2.5 Tracking

Apart from the DISCUSSES relation, in order to track changes to the PRODUCT, we need to monitor BUGS and ENHANCEMENTS. A LOG is an aggregation of LOG ENTRIES showing when a PERSON engaged in some ACTIVITY found something that needed to be changed in a PRODUCT (Figure 6).

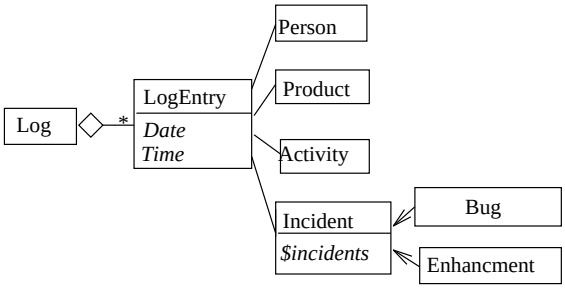

Figure 6: RA-2: ENHANCEMENT and BUG tracking.

The legal types of INCIDENTS must be predefined into $INCIDENTS variable. The RA-1 system pre-defines [17] 14 BUGS and 8 ENHANCEMENTS. An appendix to this article (§6) generalises those $INCIDENTS from their CLASS-specific RA-1 definitions to CODE-general terms.

3 Generality of the RA-2 Model

This section argues that RA-2 is general to a variety of development styles.

Consider the dashed line in Figure 5. Below that line we see some OO-specific constructs: CATEGORIES that contain CLASSES in a hierarchy and which hold METHODS; USE CASE; and SCENARIO. However, note that above that line RA-2 makes no assumptions that the PRODUCT being developed is OO. That is, at a meta-level (i.e. in the abstract classes), the repository is general to different programming methodologies. This is a desirable feature. Customising RA-2 to different methodologies should be merely a matter of adding subclasses underneath the basic RA-2 hierarchy (e.g. the ... shown in Figure 5).

Consider the difference between iterative development and the waterfall model. In the waterfall model, we only traverse the loop of Figure 4 once and there is very little feedback. Due to the iterative nature of OO development, the loop of Figure 4 may be traversed many times and there is much

feedback. In both waterfall and interative development, there is are N versions of any product. In waterfall, $N = 1$ and in iterative development, $N \geq 1$. Note that the same metrics repository can handle both approaches, as long as the model does not make the $N = 1$ version assumption. Hence the version control within RA-2:

- PLAN.REVISES
- FILE.NEXTVERSION
- SCENARIO.UPDATES

4 Other Repository Models

In this section we use the RA-2 framework to analyse other OO repository models (§4.1) and non-OO repository models (§4.2).

4.1 RA-1: An OO Metrics Repositories

The only OO repository model we are aware of is our RA-1 model [18]. RA-1 is designed for monitoring students implementing three SCENARIOS using EIFFEL over a one-semester subject. RA-1 is essentially Figure 6 and a more intricate version of Figure 2 (but without the DESIRABLE, ACCESSIBLE and COST attributes of USAGE). The SCENARIOS and PRODUCTS used for DEVELOPMENT were not exstensively modeled since they were fixed for all the student projects. However, some resource tracking occured: the students had to submit project plans in a Gantt chart (which RA-2 models in the ACTIVITIES network). The RA-1 projects are SCENARIO-driven The top-level activity in the Gantt charts were the three SCENARIOS. RA-1 also includes a reporting module for generating a set of pre-defined reports from the RA-1 repository data.

4.2 Non-OO Metrics Repositories

4.2.1 Kitchenham & Mellor

Kitchenham & Mellor [13] take a very focused view of repository modeling. They caution that:

> The danger in attempting to define a model that is *too* general is that it may be large, cumbersome, and may not yield useful measures [13, p94].

They therefore propose the minimalist repository model summarised in Figure 7 comprising of products that over some time period generate faults when some version of it is installed. The other tables of this repository are bridge entities.

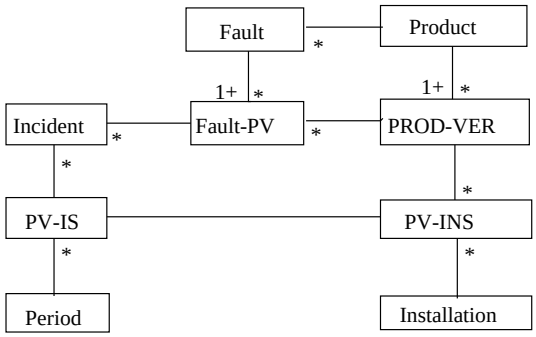

Figure 7: The Kitchenham & Mellor reliability repository model.

Their model is focused only on fault collection. The notion of a design ENHANCEMENT is missing from this repository. This model makes no attempt to record data relating to many of the issues addressed by RA-2. For example, it does not track PLANS or ACTIVITIES. Also, it does not distinguish OO from non-OO from any other development approach (e.g. by decomposing products into their CODE constituents).

4.2.2 SMART-2

SMART-2 [3] was develped as a software management tool for project management based around the US Army's Software Test and Evaluation Panel (STEP) metrics model.

In SMART-2, a phase (a.k.a. SEQUENCED ACTIVITY) is distinguished from a control task (a.k.a. SUPPORT ACTIVITY). The permitted STEP phases are: planning, requirements analysis, preliminary design, detailed design, coding, testing, sofware

integration, releasing. Further, userReqSpec, softwareReqSpec, softwareSysSpec are subclasses of requirements analysis. The permitted STEP control tasks are project mgmnt, sqa (software quality assurance), configuration mgmnt, and change resolution. SMART-2 uses numerous traceableTo links to maintain tracability between design fragments documents that generated them.

RA-2 is different from SMART-2 in several ways:

- RA-2 has a special kind of userReqSpec that is specific to OO: USE CASES.
- The RA-2 link from the USE CASE requirements DOCUMENT to PRODUCTS and TEST CASES (Figure 1) is missing in SMART-2.
- Like the Kitchenham & Mellor model, SMART-2 models BUGS but not ENHANCEMENTS.
- RA-2 makes no strict prescriptions about the valid software phases; i.e. SMART-2 is STEP-specific. RA-2 is a more general model and could model numerous process models (including STEP) using its ACTIVITY networks.
- STEP is a model of a waterfall approach. There is no version control on files or plans as offered by RA-2 (§3). Further, there is no REVIEW control task in SMART-2 suggesting that commit partitions and risk-driven development is not supported by STEP.

4.2.3 PAMPA

The PAMPA system [21] (Process Attribute Measurement and Predicting Associate) was built as part of an experiment in software support for continuous process improvement. In some respects, RA-2 is a simpler model than PAMPA:

- PAMPA has a three-level break-up of organisation, area and group, RA-2 could model the same information using nested TEAMS. Our approach allows for arbitray nested TEAMS while PAMPA assumes three-levels only.
- PAMPA specifies that files store requirements, design, documents, or source. RA-2 just assumes that all FILES are the same, but are associated to different sub-classes of PRODUCT.
- PAMPA associates softwareProducts with a supplier that can be in a reusableSourceFile or a COTSRuntFile. RA-2 has a simpler file structure and uses the single PRODUCT.VISIBILITY attribute (§2.2) to model the ability of a developer to change a program.

PAMPA permits more iteration than SMART-2. Each file-type is an aggregation of a rework class which we suspect serves the same function as our FILE.NEXTVERSION relation (Figure 5).

The PAMPA article [21] says that the softwareProduct entity somehow models resources and schedule, but is unclear on how this is done.

While we find some aspects of the PAMPA model unclear and over-specific, the major contribution of this work is the use of software agents to automatically populate parts of the metrics repository. RA-1 was developed as part of a tool to assist with the interactive logging of bugs and enhancements found in a program [18]. Experience has shown that this is a tedious task. Any automatic help with metrics collection would greatly assist the usability of a repository. Elsewhere, we are experimenting with the automatic generation of method call graphs from OO programs [4, 14]. If these experiments are successful, then we would use our message call graph generators to specify a RA-2 PRODUCT database down to the message call graph level. Such a database could be used to produce an unambiguous measure of software reuse; i.e. reuse means having the same edges in the call graph.

4.2.4 Harrison

The Harrison model [8] focuses solely on product metrics (see Figure 8). The only PRODUCT recognised is CODE PRODUCT. This repository breaks up source code down to (e.g.) (i) the level of where variables are assigned and used; and (ii) the line where procedures are defined. The Harrison model makes no reference to classes, but this could be easily added.

The Harrison work raises the question: what is the appropriate level of description for code products? Harrison argues that his detailed code-level breakdown could (i) support the standard code counting-metrics (e.g. [7]); (ii) and provide rigorous definitions of such metrics. Of the other

```
code_line(line_id, line_type).
statement(stmt_id, stmt_type).
identifier(identifier_id, indentifier_alias,
                          identifier_type).
operator(operator_id, operator_item).
proc(proc_id, proc_name).
makes_up(line_id, proc_id).
appears(line_id, stmt_id).
executed_in(operator_id, stmt_id).
used_in(identifier_id,  stmt_id).
assigned_in(identifier_id, stmt_id).
invokes(stmt_id, proc_id).
```

Figure 8: The PROLOG-based Harrison model.

product repository models studied here, only RA-1 and RA-2 approach the Harrison level of CODE detail. (recall that RA-1 and RA-2 map CODE down to the METHOD level; see Figure 5). The other models are far less-detailed: e.g. PAMPA stops at the sourceFile level; SMART-2 stops at a somewhat ill-defined module level. Note that if our experiments in method call-graph generators (§4.2.3) are successful, then we will have automatic tools that allow us to extend RA-1 and RA-2 to a Harrison-level of detail for OO programs.

4.2.5 The TAME Resource Model

The TAME resource repository model [12], as its name suggests, is focused on resource metrics. It is intended to be process independent. Resources are divided according to their type, use and description. Resource type includes the RA-2 software, hardware, and PERSON resources, plus a support resource type which we don't understand (hence, its absence from RA-2). The TAME model paper also conducts an interesting comparative review of other repository models.

The TAME repository model prompted the addition of the USAGE class in RA-2 (Figure 2). Resource planning requires knowledge of the COST and ACCESSIBILITY of that resource. Resource tracking requires a comparison of the current ACTUAL time versus the DESIRABLE ESTIMATED time.

RA-2 simplifies some aspects of the TAME model:

- The TAME model refers to a utilised and a actual attribute which we have compressed to ACTUAL.
- The TAME workType is modeled as ACTIVITY, pointInCalenderTime has become the ACTIVITY date attributes; and resourceUtilised has become the USAGE bridge class.

The TAME model also attempts to model numerous attributes relating to ESTIMATED time such as the source of the estimate (algorithm, indvidual experience, database of past projects). At the time of this writing, we are still reviewing this aspect of the TAME model.

5 Conclusion

A widely-used metrics repository model would have to be process independent, yet sympathetic to certain special constructs in different approaches. For example, the use case module of RA-2 does not extensively effect the majority of the model. That is, the use of use cases in RA-2 is optional. The other repository models reviewed here are not so sympathetic to OO (lack of iteration support and fixed names for a set of waterfall-style activities).

In terms of diagram size, the RA-2 is much bigger than some (e.g. the Kitchenham & Mellor model, the Harrison model), slightly bigger than PAMPA, somewhat smaller than SMART-2, and much smaller than some of the work reviewed in the TAME resource model paper. Our experience with students using RA-1 is that automatic support such as the PAMPA agents and out call-graph generators is required for the maintainence of the data in repository models.

Curiously, much of the RA-2 model makes little reference to OO concepts. OO software is still software and it would seem that process/resource descriptions are general to much of software. Further, Figure 5 shows us that that if we pick our meta-model with care, then at a level of abstraction, product descriptions can also be uniform across different software paradigms.

The state of the art in metrics repository design is in its infantcy. With the exception of this article and the TAME resource model paper [12], we know of no comparative analysis of repository models. Much more discussion is required before we can evolve a widely acceptable standard repository model.

6 Appendix

AL	algorithm	Improving algorithmic logic.
BL	blunder	Mental typo: code is syntactically correct but (e.g.) two variable names have been switched.
DA	data	Inappropriate function calls to a type (e.g. asking a hash table for its third element).
D1	debug, level1	Language debugger is called (development environment does not crash).
D2	debug, level 2	System locks up, crashes workstation and/or dumps core.
DO	documentation	Code should be better commented
FO	forgotten	Missing function.
IE	iteration	Defect in use of iteration (e.g. not initialising a counter, infinite loops).
IN	interface	Inappropriate function calls to methods/functions in your own application.
IV	invariant violation	Variables have somehow entered an illegal state.
MM	mismatch	A defect between specification and code.
OT	other	Something that does the fit the other categories. If lots of *others* are recorded, then the categories need extending.
SN	syntax	Error that would cause a compiler syntax error.
SS	shocking surprise	A realisation that something is basically wrong with the current design. SS is a special kind of error that prompts major design changes.
TY	typo	Typographical fatigue error.
WA	work-around	Kludge to manage some platform bug.

Figure 9: Bugs

CL	design generalisation	Desirable enhancement to existing design with an aim to future possible reuse (e.g. recognition of a useful OO design pattern, adding methods to class libraries).
CU	clean up	Change needed for enhancement or clarity;
EU	end-user usability	Making it easier for end-users to use the system.
IC	invariants checking	Ensuring that invariants remain invariant.
PF	performance	Changing a program so that it is operationally more efficient.
QU	quality	Desirable enhancement to product.
RE	reuse	Change to existing design such that application functionality can be implemented using existing code.
RO	robustness	Making code less error-prone (e.g. graceful degradation in the face of typos in user input).

Figure 10: Enhancements

References

[1] B. Boehm. A Spiral Model of Software Development and Enhancement. *Software Engineering Notes*, 11(4):22, 1986.

[2] G. Booch. *Object Solutions: Managing the Object-Oriented Project.* Addison-Wesley, 1996.

[3] C.L. Chee, S. Jarzabek, and C.V. Ramamoorthy. An Intelligent Process for Formulating and Answering Project Queries. In *SEKE '96: the Eight International Conference of Software Engineering and Knowledge Engineering*, pages 309–316, 1996.

[4] M. Connell and T.J. Menzies. Quality Metrics: Test Coverage Analysis for Smalltalk, 1996. (in press).

[5] N. Fenton, S.L. Pfleeger, and R.L. Glass. Science and Substance: A Challenge to Software Engineers. *IEEE Software*, pages 86–95, July 1994.

[6] N E Fenton. *Software Metrics.* Chapman and Hall, London, 1991.

[7] M.H. Halstead. *Elements of Software Science.* Elsevier, 1977.

[8] W. Harrison. Towards Well-Defined, Shareable Product Data. In R.W. Selby H. Dieter Rombach, V.R. Basili, editor, *International Workshop on Experiment Software Engineering: Critical Assessment and Future Directions*, pages 107–111, 1992.

[9] B. Hodgson. Personal communication. Hodgson derived his meta-model from a recent analysis of how CASE tools are used in business environments, 1996.

[10] I. Jacobson and M. Christerson. A Growing Consensus on Use Cases. *JOOP*, pages 15–19, 1995.

[11] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[12] D. R. Jeffery and V.R. Basili. Validating the TAME Resource Data Model. In *Proceedings of the 10th International Conference on Software Engineering*, pages 187–201, 1988.

[13] B. Kitchenham and P. Mellor. Data Collection and Analysis. In N. E. Fenton, editor, *Software Metrics*, chapter 6. Chapman and Hall, London, 1991.

[14] T. Menzies and P. Haynes. Empirical Observations of Class-level Encapsulation and Inheritance. Technical report, Department of Software Development, Monash University, 1996.

[15] T.J. Menzies and P. Haynes. The Methodologies of Methodologies; or, Evaluating Current Methodologies: Why and How. In *Tools Pacific '94*, pages 83–92. Prentice-Hall, 1994.

[16] D. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 5(12):1053–1058, December 1972.

[17] S Ramakrishnan and T. Menzies. An Ongoing Experiment in O-O Software Process and Product Measurements. In *Procedings SEEP'96, New Zealand*, 1996.

[18] S Ramakrishnan, T. Menzies, M. Hasslinger, P. Bok, H. McCarthy, B. Devakadadcham, and D. Moulder. On Buidling an Effective Measurement System for OO Software Process, Product and Resource Tracking. Technical Report TR96-XX, Department of Software Development, Monash University, 1996.

[19] K.S. Rubin and A. Goldberg. Object Behavior Analysis. *Communications of the ACM*, 35(9), 1992.

[20] J. Rumbaugh. Getting Started: Using Use Cases to Capture Requirements. *JOOP*, pages 8–23, 1994.

[21] D.B. Simmons, N.C. Ellis, and Kuo W. Software Process Agents. In *Proceedings of the Eigth International Conference on Software Engineering and Knowledge Engineering*, pages 323–329, 1996.

Some of the Menzies and Ramakrishnan papers can be found at *http:// www.cse.unsw.edu.au/ ~timm/pub/ docs/papersonly.html.*