A Taxonomy and Mapping of Computer-based Critiquing Tools

Norhayati Mohd Ali, John Hosking and John Grundy

Abstract—

Critics have emerged in recent times as a specific tool feature to support users in computer-mediated tasks. These computer-supported critics provide proactive guidelines or suggestions for improvement to designs, code and other digital artefacts. The concept of a critic has been adopted in various domains including: medical, programming, software engineering, design sketching and others. Critics have been shown to be an effective mechanism for providing feedback to users. We propose a new critic taxonomy based on extensive review of the critic literature. The groups and elements of our critic taxonomy are presented and explained collectively with examples, including the mapping of thirteen existing critic tools, predominantly for software engineering and programming education tasks, to the taxonomy. We believe this critic taxonomy will assist others in identifying, categorizing, developing and deploying computer-supported critics in a range of domains.

Index Terms— Design critics, critiquing systems, critic taxonomy, software tool support, survey

1. Introduction

The value of having integrated development tools, such as ArgoUML [2], MetaCase [3], Rational Rose Enterprise [4], RSSEs [5], to assist software practitioners in their development activities has received significant attention. Some of these integrated development environments have components in the form of *knowledge-based design support*, *critics*, *recommenders*, or *constraint evaluation facilities* that can advise software developers in various ways while they perform their software development tasks. Many researchers have investigated and developed such integrated support tools. However, this paper is specifically focused on research in the field of *critiquing tools and systems*.

The term "critic" was initially used [45] to describe a software program that critiques human-generated solutions. These types of tool support features, also known as critiquing systems, have evolved in recent years to help users in various computer-mediated tasks by providing feedback and suggestions for improvements [9, 12, 22, 23,24, 39, 45, 48, 53, 58, 59]. The concept of such a critic is not new and it has been accepted in various domains such as software engineering (e.g., ArgoUML [59], ABCDE-Critic [22]),programming (e.g., JavaCritiquer [53-54], RevJava [31]), medical applications (e.g., TraumaTIQ [33], ONCOCIN [cf. 57]), and others. Furthermore, reports from various studies demonstrate that a computer-supported critic is an effective mechanism for providing feedback to users. For instance, the Design Evaluator supports designers with critical effective feedback and provides reasoning about their design sketches [48]. Likewise, the Java Critiquer detects statements in student program code that can be improved for readability and to adhere to best practice [53-54]. Various *critic* definitions can be found in the literature. Though each critic tool author often provides their own definition, a common concept is that critics provide knowledge support to users who lack specific pieces of knowledge about the problem or solution domains. Thus, a critic is primarily there to detect *potential problems*; offer *advice* and *alternative solutions*; and possibly provide automated or semi-automated *design improvements* to end users. Furthermore, critic tools offer an important approach to facilitate human-computer collaborative problem solving [65]. Currently there is no accepted categorization of critics, no accepted common definition of their different features and domains of discourse, nor any framework to compare and contrast different critics and critic features in any systematic way.

Manuscript received June 2012. Revised May 2013. Accepted July 2013.

Ali is with the Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, Serdang 43400, Selangor, Malaysia
(Email: hayati@fsktm.upm.edu.my)
Hosking is with the College of Engineering and Computer Science, Australian National University, Canberra, ACT 0200, Australia
(Email john.hosking@anu.edu.au)
Grundy is with the Centre for Computing and Engineering Software Systems, Swinburne University of Technology, PO Box 218, Hawthorn, Victoria 3122, Australia. (Email: jgrundy@swin.edu.au)

To address this deficiency, this paper introduces a new taxonomy for computer-supported critics. We use the term computer-supported critics to represent the use of the critiquing approach in computer-based systems which provide critiquing functionality and interaction support to users. Computer-supported critiquing tools aim to help alleviate problems such as poor decision making, inadequate information or knowledge, human errors, and many others as mentioned in Fischer et al. [27, 28] and Silverman [60, 61]. In general, the meaning of computer-supported critics is similar to the term 'computer-based critics', as used by Fischer and Mastaglio [29]. In earlier work we developed a preliminary version of this taxonomy [6] based on our review of related literature concerning critics and which we significantly expand here. Our motivation was to use this taxonomy to assist us in designing and developing our own design critic tool for the Marama domain-specific visual language meta-tool [35]. However, it also provides a way to characterize critics and to compare and contrast a wide variety of computer-supported critic approaches, predominantly for software engineering and programming, but not limited to this domain. Our review of selected critiquing systems outside the SE domain aims to show the versatility of the approach and where lessons can be learned for new SE tool critiquing features.

In the Cambridge dictionary, a taxonomy is "a system for naming and organizing things …into groups which share similar qualities" (http://dictionary.cambridge.org). The reason for having a taxonomy is to structure an information repository for browsing. Normally, in a taxonomy, we group properties that share similar values. The aim of our paper is to present a critic taxonomy that we have developed from our reviews of the critic literature. In the following we present our motivation (Section 2) and a description of our critic taxonomy with examples (Section 3). We apply our critic taxonomy to characterize thirteen exemplar tools that have critic-support features (Section 4) and then provide comparison and discussion (Section 5). Finally we conclude with a summary of the utility and key contributions of our critic taxonomy.

2. Motivation

Information gathered from several research efforts on critics [27, 28, 36, 45, 49, 57, 61] was the initial motivation and basis for the development of our critic taxonomy. As a simple example consider a software designer manipulating a design artefact in an editing tool. The tool's critics analyse the design artefact as it changes and reveal to the designer potential problems/errors with the design artefacts e.g. wrong naming convention, over-complex design relationships, and potential misuse of design domain concepts. The critic tool offers feedback, or "critiques" the design, usually proactively as the design evolves. The tool may also suggest alternative design decisions to the designer to resolve potential problems. The interaction between designer and critic tool is iterative until the designer is satisfied with the design artefacts. Typically critic feedback is kept "unobtrusive" to the designer so as not to overly interfere with the design process.

There has been a significant amount of research in the field of critic tools, also known as critiquing systems. Previous research work and efforts, such as Fischer's endeavour [24-29], Silverman's study [60, 61], and others have attracted a wider audience to critic-based approaches. The types of support offered by these critic tools are varied. This includes problem solving and learning improvement support that have been explored by Fischer et al. [28], who has proposed cooperative problem-solving systems [28, 29] that better engage users in problem exploration. Cooperative problem-solving systems employ a critiquing approach, which allows users to design solutions for their problems rather than having solutions designed for them [28]. Fischer provides computational support for design-as-action and design-as-reflection in the critiquing tool which operationalizes Schön's notion of "talk back" with critics [28]. This supports design as a cooperative problem solving interaction between the designer and the evolving design situation [28, 29]. Fischer also claims that critics should be embedded in domain-oriented design environments (DODEs), which can then better assist designers to understand, manage, and communicate about a design problem throughout the design process [28, 29]. Prominent examples of such application were JANUS [28] and KID [46], a domain-oriented design environment for kitchen design.

In addition to the critic support provided in Fischer's work, critics also assist users to make fewer errors in their action or decision-making tasks [60]. Silverman offers criticism-based problem solving by operationalizing the judgement and decision-making (J/DM) within the critiquing approach [60]. The theory of human error and bias, and a general theory of bugs and repair strategies, was designed for expert critics [60]. Silverman [61] reports an illustrative survey of the development of several applications that were developed using the expert critiquing approach (e.g., ONCONCIN[61], ATTENDING [61], CRITTER[61], COPE[61]). The application of these expert critiquing systems in decision making problems have been shown to help reduce human errors [60, 62].

One of the most widely known and significant examples of a critic tool in the software engineering domain is ArgoUML[2, 59], an open-source Unified Modelling Language (UML) CASE tool that supports the editing of UML notation diagrams. Its critics offer suggestions to designers when a software architecture diagram violates various UML rules [59]. Similarly, the ArchStudio [1], an open-source software and systems architecture development environment that provides critics framework to support basic consistency checking and analysis of architecture description language (ADL) [21]. The LISP-Critic [24], ABCDE-Critic [22], IDEA [9] and RevJava[31], are further examples of critic-based

tools in the software design domain. These tools were developed for the domains of LISP programming, object-oriented analysis and design, design patterns and Java object-oriented software respectively.

The use of the critic concept has not to date been applied within meta-modelling tools that implement domain-specific visual language (DSVL) tools. Application has mostly been directly into application domains as evidenced above. Meta-modelling-based DSVL specification tools often employ a constraint definition/specification approach (e.g. MetaEdit+ [37], Pounamu [68], and Marama [35]) that could be used to specify critics. The process of specifying constraints for meta-modelling tools is, however, complex, requiring good knowledge of programming skills and formal approaches and involving deep cognitive load. This makes it hard for non-skilled users to understand and use such a constraint-based approach.

Inspired by existing critic tools, we have applied similar ideas to our Marama meta-modelling toolset [7, 8]. Marama is a meta-tool implemented as set of Eclipse-plugins and includes meta-tools as well as modelling tools [35]. In general, a meta-tool is a tool that allows specification and generation of another tool. Our meta-tools are used to generate complex visual modelling tools, and these modelling tools could benefit from the addition of various critics. Thus, we wanted to extend our Marama meta-tools by embedding a critic specification component. Furthermore, we wanted to assist end-user tool developers to specify and generate critics efficiently and easily for DSVL tools. Prior to achieving this, however, we needed to understand existing critic approaches to help us in designing and developing a critic specification editor for our Marama meta-tool set. We believed the critiquing approach could be useful and applicable for our Marama meta-tools though this domain is outside of the initial motivation domain espoused above. This motivated us to develop novel critic taxonomy to assist us to reason about different kinds of critics.

This paper focuses on our development of a critic taxonomy that is based on combining several approaches for critics, their definition, and framework supporting their use. This paper also shows the mapping of the critic taxonomy to thirteen existing critic tools. We classified information from the critic literature to structure our critic taxonomy. The purposes of our taxonomy are: to provide an overview of the research domain of critics (critiquing systems), to characterize the features, properties and elements included in the critic domain, and to characterize concrete critic tools (critiquing systems) and techniques within critic domain.

Critic Domain						
Critiquing Approach	Mode of Critic	Critic Rule Authoring	Critic Realisation Approach	Critic Strategy	Type of Critic Feedback	Type of critic
Comparative critiquing	Textual	Insert new critic rule	Rule-based	Active	Explanation	Completeness critics
Analytical critiquing	Graphical & 3Dimension Visualisation	Modify critic rule	Predicates	Passive	Argumentation	Consistency critics
	Multi-modal	Delete critic rule	Knowledge-based	Reactive	Suggestions	Optimization critics
		Authoring rule facility	Pattern-matching	Proactive	Examples (or precedents)	Alternative critics
		Enable/ disable critic rules	Programming code	Local	Simulation	Evolvability critics
			Object constraint language (OCL)	Global	Demonstration	Presentation critics
					Interpretations	Tool critics
					Positive	Experiential critics
					Negative	Organizational critics
					Constructive	Pattern critics
						Structure critics
						Naming critics
						Metric critics

Figure 1. Our Proposed critic taxonomy.

3. A Critic Taxonomy

Many articles and reports have been published to describe and explain critics (or critiquing systems) as an essential supporting tool for a wide range of computer users. The process of developing our critic taxonomy began by examining seminal critic related literature [27, 28, 36, 45, 49, 57, 61]. We classified the information collected from this literature into the following groups, which were tailored to meet our specific needs:

- critic domain
- critiquing approach
- modalities of critiques
- critic rules authoring
- critic realization approach
- critic strategy
- types of feedback
- critic type

Figure 1 shows the groups and elements that make up our critic taxonomy. The groupings and their elements are described in detail in the following subsections. Our aim is for the taxonomy to be applicable to computer-supported critics in general although most of our motivation, applications and examples have come from critics in the Software Engineering Critic Domain.

The first group in the critic taxonomy is the *Critic Domain*. A domain is defined as a knowledge area characterised by a group of problems with similar techniques and operational and functional specifications. Usually a domain represents a set of well-defined and coherent concepts and functions. Examples of domains include medical, business process, and education, software engineering and design environment. Critics are specified based on the domain knowledge of that particular environment/area and the use and context of critics varies from one domain to another. Thus, in order to define and specify meaningful critics for a particular context/domain, it is necessary to understand the domain knowledge being dealt with. Critics have been applied in many domains. These include critics in information systems (IS), software engineering (SE), recommender systems, education, and medicine.

For instance, a critiquing decision support system was developed by Bosansky and Lhotska [13] for healthcare specialists based on formalized medical guidelines. This system can monitor the patient's treatment progress and warn a physician in case of inconsistencies [13]. Knauss et al [39] have developed critics for requirements engineering. This illustrates the Heuristic Requirements Assistant (HeRA) editor, which offers heuristic feedback to a requirements analyst on incomplete requirements specifications. The functions of the HeRA editor are to: 1) capture high-quality requirements at the user goal level; 2) identify contradictions to other user's requirements; and 3) align user goals with the planned business process quickly [39]. Similarly, Qiu and Riesbeck [53-54] demonstrate the development of an educational critic tool, JavaCritiquer, a critiquing tool for Java programming. This tool not only supports teachers but also students. Teachers use the Java Critiquer to critique student java code whereas students obtain feedback from JavaCritiquer before sending their assignments to their teacher. Details of the HeRA and JavaCritiquertools can be found in section 4.

While critics have been broadly used in the education and medical domains, critics for recommender systems have also emerged more recently. Recommender systems represent user preferences for the purpose of suggesting items to purchase or examine [14]. Chesnevar et al. [17] explain that the aim of recommender systems is to assist users with the problem of information overload by facilitating access to essential items. McCarthy et al. [44] describe the use of a critiquing-based approach for group recommender systems. A group recommender system called **C**ollaborative **A**dvisory **T**ravel **S**ystem (CATS) is designed to assist a group of users in making decision for a vacation [44]. The CATS approach is based on a collaborative recommendation framework. An interaction component supports an individual or group interaction. A recommendation component consists of two parts: 1) an individual recommendation (the system reactively recommends cases to the user), 2) a group recommendation (the system proactively pushes recommendations to the group of users). Critics constructed by users are stored in a group user model and are used as a basis for recommendations. The contribution of CATS is to enable the user as an individual or a member of a group to interact via recommendation dialogs and to achieve consensus in their decision making about vacation planning. Figure 2 shows a screen shot of such a recommender system [44]. Several researchers have developed critiquing-based recommender systems including McCarthy et al. [43, 44], Reilly et al. [55] and Chen and Pu [15, 16]. Detailed information on critiquing-based recommender systems can be found in [15, 16].

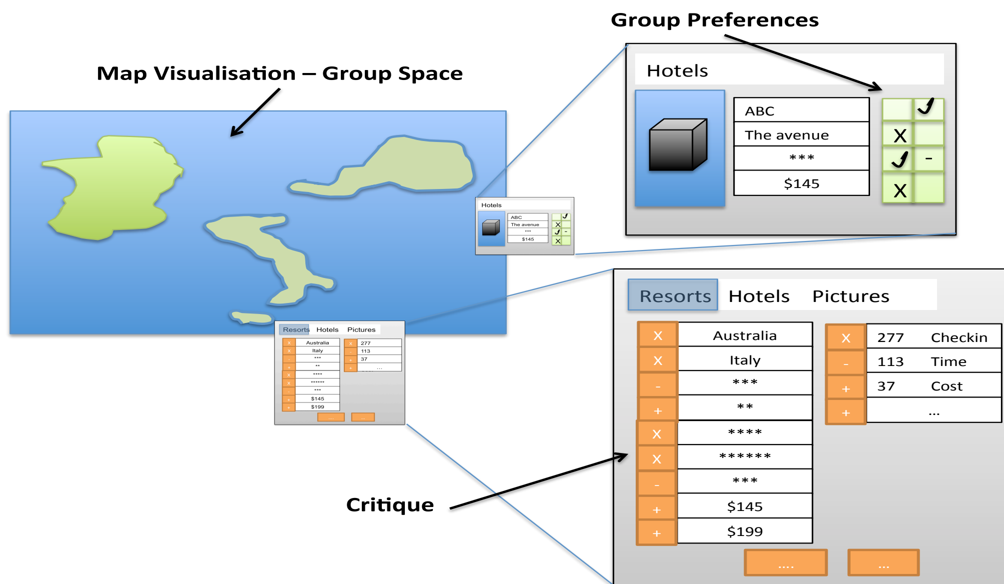

Figure 2. Example of a group recommender system.

We have briefly described a range of critic systems to demonstrate that critics are applicable to various domains and problems, and that they have often shown to be one of the more effective mechanisms of providing user feedback in a variety of computer-based design and modelling domains [45, 48, 53, 58].

The *Critic Domain* is included in our critic taxonomy as it strongly influences critic development. The other groups defined in our critic taxonomy are dependent on the domain because the type of domain knowledge they present influences the choice of elements in each group. Thus, we assume there is a strong dependency between the critic domain and the other seven groups.

3. 1. Critiquing Approach

The *Critiquing Approach* is the second group in our taxonomy and comprises *comparative* and *analytical* critiquing. Critiquing is a way to generate valid reasoning about a product or action [27-28]. Reports and articles from[27, 28, 36, 49, 53, 57, 61] have identified that critic tools commonly use comparative critiquing, analytical critiquing or both.

In a comparative critiquing approach [27, 54], complete and extensive domain knowledge is essential to generate good solutions. When a user recognizes potential problems in a design, the critic tool produces an optimal result from predefined solutions in the system. The user-proposed design is then compared with the system's solution. The comparison results in a report of the differences between the two solutions. Robbins [57] points out that a comparative approach can cause difficulties when several good solutions exist and each of the solutions are different from each other. Fisher et al. also assert that certain domains allow radically different but equally valid solutions [27]. A user also can be discouraged if the system generates its solution without recognizing the user's solution approach. As Fischer et al. point out, such a critic can only declare that the system solution accomplishes good results if the user and system's solutions differ in a fundamental way [28]. However, it cannot clarify why the user's solution is less than optimal. In a way, it hinders the exploration of different alternatives that may be good enough. In addition, Robbins also states that a comparative approach can direct users to make their work like the one that the system proposed [57]. Hence, this approach guides the user to a known solution [57]. In this approach, critic authoring is relatively intuitive and straightforward because it allows authors to write down problems and answers and the system takes care of comparison and feedback generation [53]. For example, TraumaTIQ [33] supports a physician's treatment planning. It interprets a physician's goal treatment plan, evaluates the inferred plan structure by comparing it to the system's recommended plan, and generates a critique that addresses potential problems [33]. Nakakoji [46] introduces KID (Knowing-In-Design), a domain-oriented, knowledge-based design environment for kitchen floor plan design. This tool integrates the use of a catalog base as a case library to store design artefacts. The catalog base support designers to exploit and compare precedent constructed design cases to solve problems that are similar to them.

In an analytical critiquing approach [27, 57], as long as the domain knowledge is sufficient to make judgements about a potential problem then solutions can be generated. A problem can be identified and resolved using available knowledge about the target critic domain. Hence, this approach can be applied in domains where knowledge is incomplete. In general, this approach uses rules to detect potential problems in the design and changes them into *assistance opportunities*

[57]. According to Robbins [57] an analytical critic guides the user away from recognized problems. Unlike comparative critiquing, this approach does not generate solutions on its own but instead analyses a user-proposed solution to identify any potential problems via a set of rules. Rules are typically formulated to test completeness, consistency and correctness of the target domain model.

Although an analytical approach is applicable in a broad range of domains, it is not easy to author critics for it. This is because one ideally needs to write rules for all problems in all situations [53]. Thus, the analytical approach typically involves more intensive computation compared to comparative critiquing. However, as Fischer et al. [27] state, analytical critics can be built incrementally and applied throughout the design process. According to Oh et al. [49], analytical critiquing supports exploratory problem solving better than comparative critiquing as design problems rarely have one right answer. For instance, ArgoUML is an analytical critic tool that uses analysis predicates, goals and decision type attributes to identify undesirable designs and generates feedback items with additional design context, such as contact details for relevant experts and stakeholders [58].

One critic tool that has applied both of these critiquing approaches is UIDA (User Interface Design Assistant). This critiques user interface window layouts [12]. It performs analytical critiquing by applying 72 style rules written in an OPS5-like language and comparative critiquing by recording and comparing the particular set of rules satisfied by each layout [12].

According to Irandoust [36], the choice of a critiquing approach depends largely on application domain, the characteristics of the task it supports and the cognitive support needs of the user. The key differences between these two approaches are summarized in Table 1.

Table 1. Differences between Comparative and Analytical Critiquing

Comparative critiquing	**Analytical critiquing**
• Requires a complete and extensive domain knowledge to generate a solution	• Does not require a complete domain knowledge to generate a solution
• Uses a differential analyzer [61]	• Uses rules to detect potential problems in user-proposed solution [57]
• Generates its own optimal solution, then compares it with the user-proposed solution [28]	• Critiques the user-proposed solution with respect to predefined features and effects [28]
• Guides the user to known solution [57]	• Guides the user away from the recognized problems [57]
• More suitable for well-structured domains [49]	• Can be applied to a broader range of domains [57]
• Less intrusive	• More intrusive
• Easy to author critics	• It is not easy to author critics
• Example of tools: ClassCompass [19], FFDC [50]	• Example of tools: HeRA[39], ArgoUML[59]

3. 2. Mode of Critic Feedback

The third group in our taxonomy is the *Mode of Critic Feedback*. Elements in this group comprise *textual*, *graphical and 3D visualisation*, and *multi-modal*. Presenting critic feedback, also known as *feedback* or *critiques*, is an important element to be considered in the design of any critic tool [36]. Most critics to date have provided feedback via textual messages. However, graphics can also be used for presenting critic feedback. Silverman and Mehzer [62] assert that critic feedback should be textual and visual as this usually provides the most effective results. Nakakoji et al. developed a knowledge-based colour critiquing system, eMMaC (Color Critics in Environment for Multimedia Authoring) to support novice multimedia authors in performing design of multimedia products [47]. The eMMAC system critiques the use of colour combinations, colour balance and colour associations. It critiques the use of colour in a title under construction, explains the theory underlying the critique and recommends appropriate colour usage [47]. Thus, critic designers and developers should use appropriate means wherever possible to deliver critiques. For example, matching feedback to input style can be important e.g. using in-situ text feedback if input is textual versus an intrusive dialogue, whereas using graphical feedback for graphic input e.g. colour or highlight graphical elements instead of textual message. Oh et al. [49]

recognize three modes used for presenting critic feedback in existing critic tools: text messages, graphic annotations and three-dimensional (3D) visualizations. Text messages refer to a critique presented in a written form, graphic annotation of a critique presented in a graphical form and 3D visualizations via images, or diagrams in a 3D format. We have added another element in this group, multi-modal, to cover use of animation, sound, and video to represent critiques.

Several researchers have explored the combination of textual, graphical and 3D visualizations for critique presentation in their critic tools. For instance, Oh et al. [48] developed Design Evaluator, a pen-based critic tool that generates critiques and displays them in textual and visual format. Design Evaluator covers two design domains: architectural floor plans and Web page layout design. These two design domains have different methods of displaying critiques. The Architectural Design Evaluator display critiques in three ways: as text messages, annotated drawings and texture-mapped 3D models. When a designer selects a text message critique, the tool also shows the critique in two other forms: a graphic annotation on a designer's floor plan diagram and a 3D texture-mapped VRML (Virtual Reality Model Language) model showing the path via the floor plan. The Web page Design Evaluator also generates text critiques that are linked to visual critiques via sketch annotation and design examples or cases. Similarly, de Souza et al. [22] present the **A**nnotation **B**ased **C**ooperative **D**iagram **E**ditor (ABCDE)-Critic, a system that has a construction kit to support UML class diagrams, a hypermedia system, and a critic system. In addition to textual critiques, ABCDE-Critic provides graphic annotations on UML class diagrams, such as marking in a different colour diagram elements that are detected as having errors/problems [22]. Stolze [64] developed PetriNED (**Petri N**et **ED**itor), a design environment supporting the design of Petri Nets, to show that visual critiques are possible. For example, a user constructing a Petri net model of a communication protocol may violate the 'alignment critic'. The tool notifies the user about the error by drawing lines between relevant model objects. Some critic tool researchers argue that communicating design information in a mixture of graphical critiques with text critiques is likely to be more effective than selecting only one mode of feedback [48, 62].

3. 3. Critic Rule Authoring

The fourth taxonomy group is *Critic Rule Authoring* that relates to the degree to which end users can add or modify critic rules. Elements in this group are: *insert new critic rule*, *modify critic rule, delete critic rule, enable and disable critic rule, and critic rule authoring facility*. Critic rules are one of the important components in building critics. In general, critics are composed of a single rule or groups of rules (or procedures) to evaluate different aspects of a product or design in a domain [27]. Thus, critic rules must be written for an individual product or design as well as for the critic system as a whole. According to [49], critic rules are normally written in advance by system designers to develop a critic system. In many critic systems it is often hard or impossible for a user to modify existing rules or add new critic rules after the critic system is deployed [49,53]. However, as Irandoust [36] and Oh et al. [49] have pointed out, critiquing capacity and issues may need to be adjusted from time to time. Furthermore, Fischer et al. [27] emphasize that users should not need comprehensive programming knowledge to modify critic rules. For these reasons it is important to allow users to understand critic rules and be able to modify and expand the rules by authoring new rules to incorporate in a critic system.

Riesbeck and Dobson [56] and Qiu and Riesbeck [53-54] explore the issue of authoring critic rules for an educational critic system. Riesbeck and Dobson [56] developed INDIE (Investigate and Decide), an authoring tool for intelligent interactive education and training environments. This allows users (teachers) to author and control the critic rules [56]. Qiu and Riesbeck [53-54] developed an educational critic tool for Java programming, called Java Critiquer. They explored the question of how users can author critic rules. Java Critiquer provides authoring capability, so that users (teachers) can check or modify critiques that Java Critiquer generates [53-54]. The tool allows teachers to incrementally update critic knowledge during use of the system. Tools that support customization of critic rules include ArgoUML, IDEA, Design Evaluator, and ABCDE-Critic. ArgoUML [58-59] provides a class framework, source code templates and examples to support critic implementers. Authoring a new critic requires selecting a starting template, filling in relevance and timeliness attributes, coding an analysis predicate and writing a headline and brief description [58-59]. In IDEA [9], the engineer provides new patterns and rules to select and fire new critics. Similarly, the Design Evaluator [48] allows an end-user (designer) to inspect and edit rule expressions stored in a list. ABCDE-Critic [22] also allows users to add critics, through its first-order production system. The capability of rule authoring allows end-user designers to construct and store their own critic rules [49]. A rule authoring facility allows critics to deal with various conditions and authorises end-user designers to add to the system's feedback process [49].

3. 4. Critic Realization Approach

The *Critic Realization Approach* concerns specific approaches to implementing critics. To support critic development, several approaches have been applied to the design and realization of critics. Critic implementation in various domains uses a variety of these approaches as outlined below.

3. 4. 1. Rule-based approach

Critics implemented with a *rule-based approach* consist of a condition and action, typically defined using an IF-THEN format. The **IF** part of a rule is a *condition* (also called a premise or an antecedent), which tests the truth value of a set of facts. If the *condition* is true, the **THEN** part of the rule (also called the *action*) is performed. Actions can include suggestions, explanations, argumentations, messages or precedents of problems. Rules in a rule-based approach are also known as *production rules*. A rule base stores collections of rules that have been predefined in the system. It is a statically defined set and is not modified during the execution of an application. They tend to be easy to use and to understand once implemented [67].

For instance, ABCDE-Critic [22] uses rule-based expressions to specify critics that comment on UML class diagram-based designs. The critic tool invokes critics when a condition clause is found to be true in the current design, warning a user that the design may possibly have an error [22]. Rules are coded in Java, JEOPS (*Java Embedded Object Production System*), or Prolog, according to critic type [22].

3. 4. 2. Knowledge-based approach.

In general, a critic *knowledge base* contains a set of rules and associations of compiled data, which most often take the form of IF-THEN rules (production rules). The knowledge base represents the most important component of a knowledge-based system. A knowledge base contains all true statements in a system, whether those statements are predefined rules or truths derived during the execution of the system. The format of the knowledge refers to how this knowledge is represented internally within the knowledge-based system so that it can be used in problem solving. Several knowledge representation schemes that are commonly used are: predicates, rules, frames, associative networks and objects.

FRAMER [42] enables designers to develop window-based user interfaces on Symbolics Lisp machines. FRAMER's knowledge base contains design rules to evaluate the completeness and syntactic correctness of a design as well as consistency with interface style guidelines. IDEA (Interactive Design Assistant) [9] produces design pattern critics implemented as Prolog rules directly integrated in a knowledge base. Bergenti and Poggi [9] state that IDEA's knowledge base comprises a set of design rules, corresponding critics, and a set of consolidation rules. However, creating the pattern-specific critics is not easy, requiring deep understanding of design patterns and detailed knowledge of Prolog and knowledge base structures. Robbins and Redmiles [58] point out that a knowledge-based approach is appropriate for design support where the user may lack some needed domain knowledge.

```
<lmx:pattern>
    <lmx:lhs>
      <if srcEnd="$srcEnd1;">
        <test srcBegin="$srcBegin;" srcEnd="$srcEnd;">
            <lmx:extension class="lmx.extension.SegmentMatch"/>
        </test>
      <true-case>
        <return><literal-boolean value= "true"/></return>
      <true-case>
      <false-case>
        <return><literal-boolean value= "false"/></return>
      <false-case>
    </if>
    <lmx:lhs>
      <lmx:rhs>
        <critique pos= "$srcEnd1;">
        <text>
        There is more code than you need to write. You already have a boolean value. Just
write <code>return <srcCode srcBegin= "$srcBegin;" srcEnd= "$srcEnd;"/>;</code> instead.
You never need to write an IF to return true in one case and false in the other.
        <text>
        </critique>
      </lmx:rhs>
    </lmx:pattern>

    y has a boolean value. Just write <code>return <srcCode srcBegin= "$srcBegin;" srcEnd=
"$srcEnd;"/>;</code> instead. You never need to write an IF to return true in one case and
false in the other.
      <text>
      </critique>
    </lmx:rhs>
</lmx:pattern>
```

Figure 3. Example of a Critic rule using a pattern-matching approach.

3. 4. 3. Pattern-matching approach.

According to [66], a pattern “is any arrangement of objects or entities.” A pattern matching process often involves an attempt to relate two patterns, one theoretical and another operational [66] or it can consist of left- and right-hand side rules. The most common form involves strings of characters. Many programming languages have a special string syntax to represent regular expressions (REs), which are patterns describing string characters.

For instance, the Java Critiquer tool performs automatic critiquing using a pattern matching approach [53]. When a pattern is matched, its corresponding critique is inserted below the problematic Java source code. Two types of pattern are supported: general REs and JavaML patterns. RE patterns are practical for short text segments and can be applied directly to Java source code. However, REs can become difficult[53]. Thus, a built-in pattern editor is provided to support teachers in incremental authoring of patterns, which is more direct and simpler than editing REs. Critic rules in the Java Critiquer are written in an XML format, LMX (language for Mapping XML). The left-hand side of a rule is a LMX pattern and the right-hand side is a critique. The “pattern matcher” matches the rule patterns against the JavaML code, and returns a list of triggered critiques [53]. Figure 3 is an example of a critic rule written using this pattern matching approach.

3. 4. 4. Predicate logic

According to Tyugu [67], predicate logic is based on the idea that “sentences (propositions) really express relationships between objects as well as qualities and attributes of such objects (can be people, other physical objects, or concepts).” Such relationships or attributes are called predicates. The objects are arguments or terms of the predicate. The use of terms allows a predicate to express a relationship about many different objects rather than just a simple object [67]. Using predicates more complex statements about the world can be made than with propositions. Predicates can also be used to represent an action or an action relationship between two objects [67].

One example of a critic tool that applies a predicate approach is the Design Evaluator [48]. This contains three layers: *Description*, *Evaluation*, and *Visualization*. The *Evaluation* layer evaluates sketches with predicates that embody design rules. The tool compares the recognized spatial information with each rule. If it finds a rule violation, it generates a design critique displayed in the *Visualization* layer [48]. In the *Evaluation* layer, rules are coded as Lisp predicates that apply to the design objects. The rule expressions are stored in a list that the end user (designer) can inspect and edit. Figure 4 shows an example of a rule for an architectural floor plan domain [48].

- Rule statement: A ward be no smaller than 10,000 area units

  A minimum area rule: express a minimum area requirement about a specific room.
  (<Minimum-area><room><minimum-size>)

  (MINIMUM-AREA WARD 10000)

- Rule Statement: typical room placement in hospital design that states ER, TRIAGE, CLINICAL-FOR-OUTPATIENT, and DAYWARD should be placed in the CLINICAL-ZONE

  A room placement rule: all rooms in the list inside the inner parentheses should be in (or not in) the given zone.
  (<Placement-rule>
  <Zone>(<Room><Room><Room>...))

Figure 4. Example of rules for an architectural floor plan using a predicate style.

3. 4. 5. Object constraint language (OCL) expressions

According to [38], the Object Constraint Language (OCL) offers a means to specify accurately the semantics of an object-oriented model. These are expressed in invariants and pre-and-post conditions, which are different types of constraint [38]. OCL can be used to construct logical expressions that access attributes, invoke operations, navigate along associations, and manipulate collections [20]. An analysis of model checking by [10] demonstrates the use of OCL to express constraints in a simple domain-specific language (DSL) called Class Diagrams (CD). They argue that OCL needs extensions to support the severity of a constraint, i.e. a representation of the degree of flaw in a problem, which can be classified either as an *error*, a *warning* or a *critic* [10]. In their CD example, they show how a critic is expressed using an OCL expression. Figure 5 shows an example of critics written using OCL expressions [10].

- Critic statement: the name of Classifier must be unique within its package

  OCL expression:

  ```
  Context Classifier
  Inv: not self.package.contents->exists (e|(e <> self) and (e.name = self.name))
  ```

- Critic statement: the name of a Classifier should begin with an upper case letter.

  OCL expression:

  ```
  Context Classifier
  Inv: not (let firstChar: String = self.name.substring(1,1) in firstChar<>firstChar.toUpper())
  ```

Figure 5. Example of Critics written using OCL expressions.

3. 4. 6. Programming Code

Critics are often designed and realised via programming code. For instance, critics in ArgoUML [59] are coded as subclasses of the Java class Critic. Critic defines several methods that may be overridden to define and customize a new critic. Each critic's constructor specifies the headline, problem description, and relevant decision categories. The central method is a predicate that accepts a design element to be critiqued and returns true if a problem is found [59]. RevJava [31] also implements critics via programming code, i.e. Java class files. The tool is used to analyse and critique object oriented software.

3. 5. Critic Strategy

The *Critic Strategy* classifies critics by several strategic dimensions. These are based on Fischer's suggestions [25]. Irandoust [36], Oh et al. [49], Qiu and Riesbeck [53], and Robbins [57] also support Fischer's proposed dimensions. Our taxonomy's critic strategy categories adhere to Fischer's critic dimensions as shown in Table 2.

Table 2. Critic Strategies.

Critic Strategy	Brief Description
Active critics	Continuously critique the user's design/work
Passive critics	Wait until the user asks for a critique
Reactive critics	Critique the design/work that the user has done
Proactive critics	Guide user by presenting guidelines before the user makes a decision
Local critics	Critics that evaluate individual design elements
Global critics	Critics that consider interactions between most or all of the elements in a design

In critic development, a critic developer must consider using active critics, passive critics or both. An *active critic* [25] continuously monitors user's tasks, warns the user as soon as a critic rule is violated and offers critic feedback (a critique). An active critic makes users aware of their unsatisfactory design/work when the potential problem is easy to correct. However some users may find it a distraction to be continuously critiqued without having a chance to develop their own design/work and corrections. In contrast, a *passive critic* [25] only works when a user asks for a check of critic rule violation. In this scenario, after the user completes preliminary design/work, the user asks for evaluation of the design/work. Passive critics are less intrusive than active critics as they allow the user to control when to activate them. The problem with passive critics is that most of the time, the user does not activate them early enough to prevent potential problems [53]. Fischer [25] remarks that active critics are suitable for guiding novice users and passive critics are better for intermediate users.

ArgoUML provides active critics when a user attempts to draw a design diagram. For example, when a user selects a new class to place in a class diagram design, several critics fire to indicate that part of the design has been started, but not yet finished. Java Critiquer uses passive critics because as Qiu and Riesbeck [53] state, it is not a requirement to avoid students from making mistakes. Thus, Java Critiquer provides an opportunity for learning and allows students to concentrate on their programming tasks without interruption [53].

Reactive and *proactive* critics provide another critic dimension perspective. A *reactive* critic [25] provides critiques on the user's accomplished design/work, whereas a *proactive* critic attempts to lead the user before the user makes a specific decision. Similar to these two critics are the critic dimensions suggested by Silverman [61]: *before*, *during* and *after*.

Silverman's *before critic* is similar to Fischer's proactive critic. *During* and *after critics* can be viewed as Fischer's reactive critics. However, *during* and *after* critics differ in terms of whether a user's work is completed or not. SEDAR [32] adopts Silverman's dimensions and uses all three strategies: *before* (error prevention), *during* (design review critic, design decision) and *after* (error detection). HeRA [39] provides proactive support; while users type requirements, it analyses the input and warns of detected ambiguities or incompleteness.

Finally, critics can be classified as either *local* or *global*. Local critics [25] evaluate individual design elements and global critics [25] involve interactions between most or all of the elements in a design. HeRA [39] provides users with both local and global critics. According to [39], the local critics are concerned with the current focus of the requirements editor (i.e. requirements, use cases, and a glossary), while global critics allow users to analyse a global perspective in terms of a list of all critiques and inference across global process diagrams (i.e. UML Use Cases, Event-driven Process Chain models, and Use Case Point View).

3. 6. Type of Critic Feedback

The next taxonomy group is the *Type of Critic Feedback*. There are ten elements in this group: *explanation*, *argumentation*, *suggestion*, *example* (or precedent), *interpretation*, *simulation*, *demonstration*, *positive feedback*, *negative feedback*, and *constructive feedback*. There are many ways to present critic feedback, also known more briefly as feedback, in a critic tool [36]. Oh et al. [49] describe the types of critic feedback as one aspect of the critic's intervention techniques. Critic tools can offer critic feedback to users by choosing the appropriate techniques from the ten elements. However, the most widely used techniques are explanation, suggestion, and argumentation.

The *explanations* technique is widely used. Explanation, as defined in the Cambridge dictionary, is "*details or reasons that someone gives to make something clear or easy to understand*". Thus, the critiques provided by a critic give explanations so that a user has the chance to assess the details and reasons behind the critique before making a decision as whether to accept or act on the generated critique. Explanations can be focused on violations of general guidelines or the differences between the user's design solution and the system's solution [28]. An explanation facility is also needed to show the correctness and usefulness of the critic tool's recommendation. It is essential to validate a critique via explanation as, without valid details or reasons, a user will generally not accept it [36]. Explanations provided can be simple or in-depth. A simple explanation component normally provides pre-stored text explanations. In detailed explanations, hypertext techniques have been shown to be very efficient for providing contextualized explanations [27]. Fischer and colleagues contribute the incorporation of hypertext into critic's feedback loop and the creation of what they call "minimalist explanation" [29]. Via hypertext links, the user can obtain more in-depth explanations. Explanations too can be represented textually visually or both.

Argumentation is another option for critic feedback. This mechanism for explanation can contain issues, answers, and arguments about a product or design domain. A user, not understanding critiques offered by a critic tool, may want to have more information about them. Via an argumentation component, the user can obtain the required information to justify the critique. Examples of critic tools that provide an argumentation style are Indie [56], ABCDE-Critic [22] and HeRA [39]. These tools are developed for the domains of education learning, object-oriented analysis and design, and requirements engineering.

Indie (Investigation and Decide) is an authoring tool for intelligent interactive education and training environments. The tool helps authors (i.e. teachers) to create knowledge bases for critiquing student arguments. A student's argument is compared against the argument model via the Indie *Critiquer* modules. One of the knowledge bases in the Indie tool has argument models to describe what makes good and bad arguments. The argument contains a claim about a scenario, and a set of evidence. ABCDE-Critic [22] incorporates an *argumentative hypermedia system* to provide in-depth explanation for users that don't understand or want more information about critics. This comprises issues, answers and arguments about the design domain. Likewise, HeRA [30] facilitates its computer-based critiques via an argumentation component, allowing users to adhere to warnings or argue against them.

Some critics offer *suggestions* to change a user's solution. A suggestion style critic, also known as *solution-generating critic* [27] is capable of suggesting alternatives to the user's solution. An example is JANUS [27], a kitchen design system, where a simple problem-detecting critic points out that there is a stove close to a door. Another option is to provide examples (precedents) to support critics. Examples are a way of helping users to understand something by showing them how it is used. For example, the Design Evaluator [48] provides an exemplar Web page for the designer to look at when a critique is selected.

Another option for presenting critic feedback is to provide *positive* or *negative feedback*. Positive feedback praises a user when a good design/solution is produced. Negative feedback complains when a user produces a poor design/solution. Positive and negative feedback is related to how humans make decisions as humans tend to evaluate based on advantages and disadvantages, pros and cons. In PetriNED [51], positive critiques are delivered in a graphical way, close to the user's focus of attention. This is helpful to those users who are interested in obtaining positive feedback.

Apart from the styles stated above, critic feedback can be presented through use of simulation, demonstration, interpretation and constructive feedback. For instance, a *simulation* component in HeRA tool provides 'what-if' analysis and derives three models (i.e. *UML Use Case Diagrams*, *EPC Business Processes* and *Use Case Points Estimations*) while a use case is written [39]. The FFDC [50] offers an *interpretation* component to interpret students' design solutions. It uses a *demonstration* component to present graphical annotations in a Construction Interface and displays a written statement to describe the detected condition. Likewise, ArgoUML [59] provides *constructive feedback* to support design improvements by means of 'To do' list, design checklist and wizards. A mixture of styles in presenting critiques facilitates users to clarify their understanding and improve their knowledge.

3. 7. Critic Type

The last group in our taxonomy is the *Type of Critic*. Critics can be classified according to the type of domain knowledge that they present [57, 58]. Thus, the Critic Domain and Type of Critic complement each other. Table 3 shows a list of critic types we define in our taxonomy based on [9], [19] and [58]. According to Robbins [57] critic types are descriptive rather than definitive. In fact, new categories can be defined based on the application domain. For instance, IDEA [9] offers pattern-specific critiques to assist architects in finding and improving the realisations of design patterns in UML designs. Similarly, [19] defines three categories of critic: structure, naming and metric critics for the ClassCompass tool.

Table 3. Critic Types

CRITIC TYPE	DESCRIPTION
Correctness critics	detect syntactic and semantic flaws [58]
Completeness critics	remind the designer to complete design tasks [58]
Consistency critics	point out contradictions within the design [58]
Optimization critics	suggest better values for design parameters [58]
Alternative critics	prompt the architect to consider alternatives to a given design decision [58]
Evolvabilitycritics	address issues such as modularization, that affect the effort needed to change the design over time [58]
Presentation critics	look for awkward use of notation that reduces readability [58]
Tool critics	inform the designer of the available design tools at the times when those tools are useful [58]
Experiential critics	provide reminders of past experiences with similar designs or design elements [58]
Organizational critics	express the interest of other stakeholders in the development organisation [58]
Pattern critics	improve a design via design patterns [9]
Structure critics	detect problems that involves structural properties [19]
Naming critics	identify potential sources of confusion introduced by names [19]
Metric critics	report when the number of occurrences of some aspect of a design is beyond normal values [19]

4. Applying the Taxonomy

In this section, we apply our new critic taxonomy to position a variety of critic tools within the computer-supported critic domain. A number of representative systems and tools that adopt or implement the critic concept have been selected as illustrative, including a mix of research prototype and open source tools. As discussed earlier, we use predominantly SE and programming tools to illustrate our taxonomy, but also a few exemplars outside the SE domain. The later are used to show the way critics can be used in a variety of contexts. We also show that the concept of the critic and our critic taxonomy dimensions can be applied at multiple levels in SE tools: programming statements, as in JavaCritiquer, design models, as in ArgoUML, and requirements and domain models in MaramaCritic.

Successful critic implementations in the past have guided others to develop computer-supported critics with the intention of enhancing the performance of large numbers of users in performing their computer-mediated tasks. There are too many critic based systems to comprehensively cite; the review of critiquing tools in this paper should thus be considered expressive and representative of the system domain. Most critiquing tools developed in the 80s and 90shave been re-

viewed by multiple researchers such as Silverman [61], Fischer et al. [28], and Robbins [57]. For the purposes of testing the application of our taxonomy we thus decided to search and select a representative range of critiquing tools developed within the more recent decade of 2000-2010. We also focused on research prototype tools rather than commercial critic tools because we were not interested in the 'industrial hardening' issues of usability and scalability. Commercial IDEs such as Eclipse and Visual Studio, and commercial software design tools such as Enterprise Architect, have regularised many of the concepts of critics. However, they typically support only some aspects of our critic taxonomy. Specifically, their support for critic authoring and tailoring is usually limited, feedback is still predominantly textual, and realisation approaches are usually rule-based or pattern matching.

Critic domain is the first group defined in the critic taxonomy. To illustrate the critic taxonomy application, we examine several critic tools introduced within the decade range 2000-2010, as well as with our own research tool, MaramaCritic. These are mainly from the software engineering and education domains. We include both research prototypes and widely used commercial tools. In total we examine seven tools from the software engineering (SE) domain, five tools from the education domain and one tool from the design engineering/architecture domain, with a summary description of each shown in Table 4. We apply the taxonomy of Table 4 to this set of tools, illustrating the resulting taxonomic categorisations using a compressed format, as shown in Figure 6.

1.Critic Domain:						
2.Critiquing Approach	**3.Mode of Critic Feedback**	**4.Critic Rule Authoring**	**5.Critic Realisation Approach**	**6.Critic Strategy**	**7.Type of Critic Feedback**	**8.Type of Critic**
C	T	IR	RB	Ave	E	Co
A	G&3D	MR	KB	Pve	Ar	Com
	MM	DR	Pr	Rve	Sug	Con
		E/DR	PM	Pro	Ex	Opt
		AR	OCL	Lo	Sim	Alt
			PC	Gl	Dem	Evo
					Int	Pre
					PosF	Tool
					NegF	Exp
					ConF	Org
						Pat
						Str
						Nam
						Met

Critiquing Approach	Mode of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Strategy	Type of Critic Feedback	Type of Critic
C=Comparative, A= Analytical	T=Textual, G&3D=Graphical & 3-Dimension, MM=Multi-Modal	IR=Insert Rule, MR=Modify Rule, DR=Delete Rule, E/DR= Enable/ Disable Rule, AR= Author Rule	RB=Rule-Based, KB=Knowledge-Based, Pr=Predicates, PM=Pattern-Matching, OCL=Object Constraint Language, PC= Programming Code	Ave=Active, Pve=Passive, Rve=Reactive, Pro=Proactive, Lo=Local, Gl=Global	E=Explanation, Ar=Argument, Sug=Suggestion, Ex=Examples, Sim=Simulation, Dem=Demonstration, Int=Intepretation, PosF=Positive Feedback, NegF= Negative Feedback, ConF=Constructive Feedback	Co=Correctnes, Com=Completeness, Con=Consistency, Opt=Optimisation, Alt=Alternative, Evo=Evolvability, Pre=Presentation, Tool, Exp= Experience, Org= Organisation, Pat = Design Pattern, Str= Structure, Nam= Naming, Met= Metric

Figure 6. Our Critic taxonomy in a compressed style (top) with legend (bottom)

4. 1. ArgoUML

ArgoUML is an object-oriented design tool using the Unified Modelling Language (UML) design notation. It is a *design critic* tool that supports several identified cognitive needs of software designers. Figure 7 shows the ArgoUML user interface. As Robbins and Redmiles state, "design critics are agents that check the design for potential problems" [59]. Thus, ArgoUML has predefined agents, called *critics* that constantly check the current model designed by software designers. A critic will generate a *ToDo Item* (as a critic feedback item or a critique) in the *ToDo* list if the conditions for triggering a critic occur.

The *ToDo Item* (as shown in Figure 7) is presented in a constructive manner which is helpful to software designers as it contains an explanation of the problem, some suggestions about how to resolve the problem, and if one exists, a wizard to assist the designer to resolve it automatically. A *ToDo* item generated by a critic remains in the *ToDO* list until the cause of the problem is removed either manually by the designer or by following the actions suggested by the tool's wizard.

Table 4. Thirteen surveyed critic tools/systems.

Tool Name (year-based on published paper)	Description	Application Domain	Application Type
ArgoUML (2000)	"Critiquing is done continuously and designers need not request that critics be applied or even know that any particular critic exists." [59]	Software engineering (UML designs)	Research and open source tool
ABCDE-Critic (2000)	"…implements a construction kit supporting UML class diagrams, an argumentative hypermedia system, and a critic system, where the user is able to define his own critics." [22]	Software engineering (Class diagram design)	Research prototype tool
IDEA (2000)	"…is a critiquing system that we developed to work in direct interaction with the software architect to propose pattern-specific critiques." [9]	Software engineering (design patterns)	Research prototype tool
RevJava (2002)	"it is used to analyse and critique object oriented software." [31]	Software engineering (object oriented Java)	Research prototype tool
DAISY (2003)	"…a critiquing system is able to check the consistency of models created during domain and application engineering." [23]	Software engineering (software modelling)	Research prototype tool
Submit! (2003)	"..aims to enhance teaching and learning in computing by developing automated web-based tools that assist in providing critical feedback to students about the computer programs they write" [51]	Education (program critiquing)	Research prototype tool
JavaCritiquer (2003)	"a critiquing system to teach students how to write clean, maintainable and efficient code." [54]	Education (Java programming)	Research prototype tool
Design Evaluator (2004)	"is a pen-based system that provides designers with critical feedback on their sketches in various visual forms." [48]	Design engineering (design sketching)	Research prototype tool
ClassCompass (2007)	"an automated software design critique system with critics that comment on high-level design issues rather than diagram completeness." [19]	Education (software design)	Research prototype tool
Essay Critiquing System (2009)	"is a system that has incorporated LSA to provide immediate feedback on writing ideas to students on essay topic by comparing student and model essays" [41]	Education (essay critic)	Research prototype tool
FFDC (2009)	"…as a step toward creating computer-based critics that support design learning in studio setting." [50]	Education (architecture design)	Research prototype tool
HeRA (2009)	" a feedback centric requirements editor to help analysts to control the information overload." [39]	Software engineering (requirements engineering)	Research prototype tool
MaramaCritic (2010)	a generic critic specification component to assist end-user tool developers to specify and generate critics efficiently and easily for domain-specific visual language (DSVL) tools.	Software engineering (DSVL tool)	Research prototype tool

The design feedback in ArgoUML also provides contact information (email address) for relevant experts and stakeholders to assists designers to resolve the problem at hand. ArgoUML's *ToDo* list is practical as it reduces the designer's reliance on short-term memory and offers convenient ways to organise and browse items. The critics in ArgoUML are not intrusive as the user can disregard them completely or disable one or all of them via the critics' configuration menu. Critics in ArgoUML are not user defined as they are implemented as Java classes compiled as part of the tool. However it does provides a class framework, source code templates and examples to facilitate the critic implementation process [59]. Thus, adding new critics can be done by modifying the source code requiring Java expertise. Details of ArgoUML can be found at [2].

Figure 8 shows the mapping of the ArgoUML tool to our critic taxonomy. Items in blue represent elements supported by the tool. To summarise this analysis: ArgoUML is a software engineering - specifically UML diagramming - domain critic, and its critiquing approach is analytical. It provides textual and graphical modes of critic feedback. It supports inserting, modifying and editing critic rules. It uses predicate and programming language critic realisation approaches, and critic dimensions include active, passive and local. Types of critic feedback include explanation, argumentation, suggestion and constructive feedback, and types of critic include correctness, completeness, consistency, optimization, alternative, evolvability, presentation, tool, experiential, organizational and pattern (Singleton pattern only).

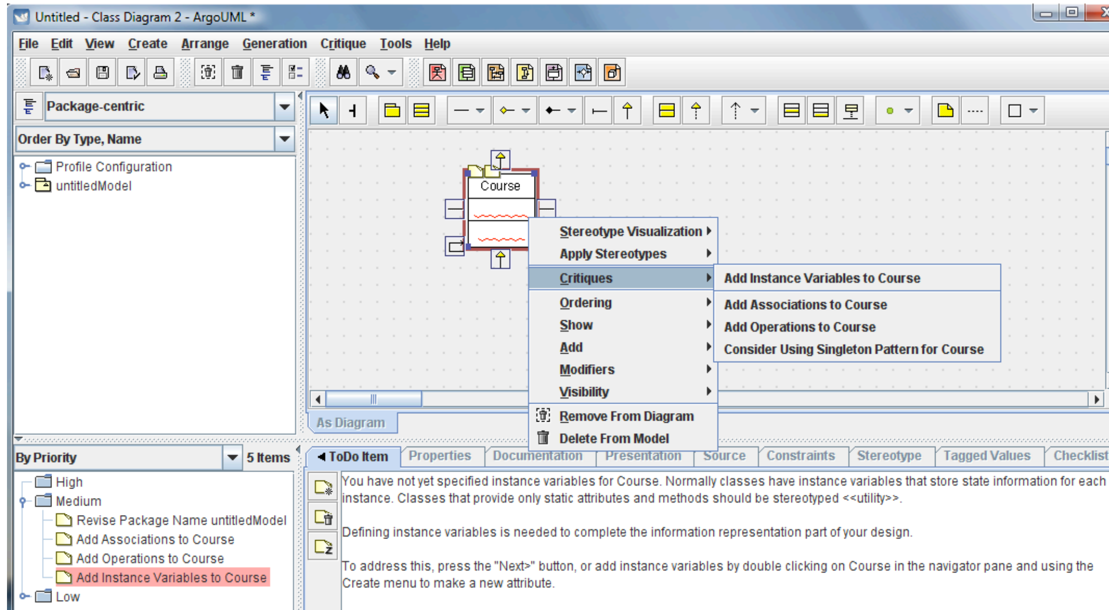

Figure 7.Screenshot of the ArgoUML user interface.

			Critic Domain:	Software Engineering	(UML Designs)	
Critiquing Approach	Modes of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Strategy	Types of Critic Feedback	Types of Critic
C	T	IR	RB	Ave	E	Co
A	G&3D	MR	KB	Pve	Ar	Com
	MM	DR	Pr	Rve	Sug	Con
		E/DR	PM	Pro	Ex	Opt
		AR	OCL	Lo	Sim	Alt
			PC	Gl	Dem	Evo
					Int	Pre
					PosF	Tool
					NegF	Exp
					ConF	Org
						Pat
						Str
						Nam
						Met

Figure 8. The mapping of the ArgoUML tool to the critic taxonomy.

4. 2. ABCDE-Critic

ABCDE-Critic [22], a Domain Oriented Design Environment (DODE) for object-oriented analysis and design, uses critics to check UML class diagrams. The environment implements a construction kit supporting UML class diagrams, an argumentative hypermedia system, and a critic system. It uses rule-based expressions to specify critics that comment on UML class diagram-based designs. The critic system fires critics when condition clauses are found to be true in the current design warning the designer that the design may have a problem/error. Critic properties in ABCDE-Critic are: 1) critic name, 2) state (active, passive, disabled), 3) a quick critic explanation, 4) an argumentation providing more in-depth explanation, 5) critic importance, 6) a set of rules, and 7) a set of solutions.

Critic feedback is presented as annotations attached to diagram elements that trigger the critic. The annotations are shared with all other designers who are owners of these diagram elements. Critic feedback is displayed in two views. The first is in the pop up "*Things to take care of*" window. This displays the critic name and quick explanation in a list box. The second is where the annotations created for the diagrams being constructed are displayed in the graphics interface

component known as the *annotation column*. ABCDE-Critic uses a Design Rationale (DR) model to record the justification behind the design decision made during object-oriented analysis and design activities. Designers can define and control the critic's state (active, passive, disable) as necessary. ABCDE-Critic allows designers to add critics to the tool via a first-order production system.

ABCDE-Critic offers an automatic critiquing capability, which is similar to ArgoUML's. Although the critiquing domain is similar to the ArgoUML, the scope of the critiques differs. ArgoUML critics generate critiques that support UML class diagrams, state diagrams, use case diagrams, activity diagrams and collaboration diagrams. ABCDE-Critic provides a UML class diagram editor that focuses only on class diagram critics. ABCDE-Critic provides a different approach to ArgoUML in that it supports cooperation among designers via manual critiquing, warning all designers involved in the problem. It also allows other designers to add alternatives to the set solution of a critic. Thus, designers can communicate with the critiquing system as a "design partner". ArgoUML does not provide support for manual and group critiquing. Figure 9 shows the mapping of the ABCDE-Critic tool to our critic taxonomy. Compared to the ArgoUML characterisation in Figure 8, ABCDE-Critic supports richer critic authoring than ArgoUML; a different critic realisation approach; and proactive critics. It does not provide constructive feedback.

			Critic Domain:	Software Engineering	(UML Class Diagrams)	
Critiquing Approach	**Modes of Critic Feedback**	**Critic Rule Authoring**	**Critic Realisation Approach**	**Critic Dimension**	**Types of Critic Feedback**	**Types of Critic**
C	**T**	**IR**	**RB**	**Ave**	**E**	**Co**
A	**G&3D**	**MR**	KB	**Pve**	**Ar**	**Com**
	MM	**DR**	Pr	Rve	**Sug**	Con
		E/DR	PM	**Pro**	Ex	Opt
		AR	OCL	**Lo**	Sim	Alt
			PC	Gl	Dem	Evo
					Int	Pre
					PosF	Tool
					NegF	Exp
					ConF	Org
						Pat
						Str
						Nam
						Met

Figure 9. The mapping of the ABCDE-Critic tool to the critic taxonomy.

4. 3. IDEA

Interactive **De**sign **A**ssistant (**IDEA**) interacts with software architects to propose pattern-specific critiques [9]. IDEA is designed to automate the task of finding realisations of design patterns used in UML diagrams and then improving the diagrams. The improvement of the design is made through critiques that are presented to software architects. IDEA uses design pattern-based critics implemented as Prolog rules directly integrated into a knowledge base. The IDEA approach is that a UML design under construction is analyzed to detect all pattern realisations. If a pattern is detected then it is called *detectable*, otherwise it is called *undetectable* because of incomplete information on the diagrams. When a pattern realisation is discovered, IDEA examines pattern-specific rules to select a set of critics to improve the design realisation.

IDEA provides the architect with two lists, the "pattern list" and the "to-do list". The "pattern list" contains all patterns that IDEA found in the UML model. There are eleven patterns detected by IDEA: Template Method, Proxy, Adapter, Bridge, Composite, Decorator, Factory Method, Abstract Factory, Iterator, Observer and Prototype. The "to-do list" (the critic feedback) is the list of all selected critiques organized by their importance (high, medium, and low). IDEA allows architects to control pattern detection directly through these lists.

The knowledge base of IDEA comprises a set of design rules, corresponding critics, and a set of consolidation rules. These are maintained dynamically as patterns and rules can be added and removed when required. However, the rules for creating the pattern-specific critics are not easy to understand or author as this requires a high-level of understanding of a design patterns and detailed knowledge of Prolog and knowledge base structures. IDEA provides support for automatic critiques of UML diagrams that are similar to ArgoUML and ABCDE-Critic. The nature of the critiques produced by IDEA is different from the previous tools as it focuses on design-pattern critics. However, IDEA and ABCDE-Critic allow their users to add and delete critics when necessary and ArgoUML does not offer these features. Figure 10 shows the mapping of the IDEA tool to our critic taxonomy.

			Critic Domain:	Software Engineering	(Design pattern)	
Critiquing Approach	Modes of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Dimension	Types of Critic Feedback	Types of Critic
C	**T**	**IR**	**RB**	Ave	**E**	Co
A	**G&3D**	**MR**	**KB**	Pve	Ar	Com
	MM	**DR**	Pr	Rve	**Sug**	Con
		E/DR	PM	**Pro**	Ex	Opt
		AR	OCL	**Lo**	Sim	Alt
			PC	Gl	Dem	Evo
					Int	Pre
					PosF	Tool
					NegF	Exp
					ConF	Org
						Pat
						Str
						Nam
						Met

Figure 10. The mapping of the IDEA tool to the critic taxonomy.

4. 4. DAISY

Following their work on ABCDE-Critic, de Souza et al. [23] developed **DAISY** (**D**omain and **A**pplication engineering using **I**ntegrated critiquing **SY**stems), which supports the construction of domain engineering and application engineering models. The goal of their approach is to support consistency management in these models [23]. Domain engineering comprises three main activities: 1) domain analysis, 2) domain design, and 3) domain implementation. Their work is more focused on diagrams and models created during domain analysis and domain design. Application engineering complements these processes, producing software based on the domain engineering process.

DAISY is built on top of ABCDE-Critic. It supports consistency checking of models through three different critic systems. The first assists the development of feature diagrams via seven different critics. The feature diagrams show the architectural structure of software features. In this work, DAISY deals with software architecture diagrams and class diagrams. The second is used during application engineering to assess UML class diagrams using object-oriented design heuristics via approximately twenty critics. These two critic systems are used to improve the overall quality of the UML models. The third system detects potential inconsistencies and other errors that might occur in the mapping between domain and application models. Seven critics are used here. The contribution of DAISY is inconsistency detection in a software engineering models via use of three critic systems. Though the number of critics implemented is small they could be further extended. DAISY is similar to the three previous tools because these tools all support critics for editing UML diagrams and identifying common design errors made by users. Feature diagrams and inconsistency detection are the main elements that distinguish DAISY from the three previous tools as these elements are not offered by the other tools. Figure 11 shows the mapping of the DAISY tool to our critic taxonomy.

			Critic Domain:	Software Engineering	(feature diagrams and class diagrams)	
Critiquing Approach	Modes of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Dimension	Types of Critic Feedback	Types of Critic
C	**T**	**IR**	**RB**	**Ave**	**E**	**Co**
A	**G&3D**	**MR**	KB	**Pve**	**Ar**	**Com**
	MM	**DR**	Pr	Rve	**Sug**	**Con**
		E/DR	PM	Pro	Ex	Opt
		AR	OCL	**Lo**	Sim	Alt
			PC	**Gl**	Dem	Evo
					Int	Pre
					PosF	Tool
					NegF	Exp
					ConF	Org
						Pat
						Str
						Nam
						Met

Figure 11. The mapping of the DAISY tool to the critic taxonomy.

4. 5. Submit!

Submit! is a web-based system for automatic program critiquing [51]. Submit! provides critical feedback to students about their programming work. Pisan et al., [51] claim that the aim of Submit! is to improve the monitoring of student progress and to provide formative assessment that supports self-directed learning. Students are allowed to employ the critiquing tool to correct their own mistakes before final submission of an assignment. The main functions of the system

includes acceptance of submissions from students in the form of files containing program code, analysis of them according to specific criteria, and return of an analysis report to the student [51].

Initially the critiquing process only deals with a simple critiquer that runs the program on a standard dataset. If the program does not generate the expected output, feedback will be given to the student. Advanced critics are subsequently incorporated into the Submit! tool using an incremental approach. Although any type of critic can be added, Submit! mostly works with correctness critics. A preliminary study of the impact of Submit! on student results indicates that students who use the system to obtain feedback on assignment submissions do better than those who do not use it [51]. The mapping of Submit! to our critic taxonomy is shown in Figure 12. In confirming our mapping, Pisan, one of the Submit! authors, advises that each "critic" is a separate program written by an instructor and all uploaded critics are executed on student submissions [52].He also notes that a critic can be “archived”, which has a similar effect to disabling it. Apart from that, editing a critic usually requires downloading the code, modifying it, and then uploading the code again, with often no support from the system [52].

			Critic Domain:	Education	(computer programming)	
Critiquing Approach	**Modes of Critic Feedback**	**Critic Rule Authoring**	**Critic Realisation Approach**	**Critic Dimension**	**Types of Critic Feedback**	**Types of Critic**
C	T	IR	RB	Ave	E	Co
A	G&3D	MR	KB	Pve	Ar	Com
	MM	DR	Pr	Rve	Sug	Con
		E/DR	PM	Pro	Ex	Opt
		AR	OCL	Lo	Sim	Alt
			PC	Gl	Dem	Evo
					Int	Pre
					PosF	Tool
					NegF	Exp
					ConF	Org
						Pat
						Str
						Nam
						Met

Figure 12. The mapping of the Submit! tool to the critic taxonomy.

4. 6. RevJava

RevJava [31] is a tool used to analyse and critique object-oriented software. According to Florijn [31], the Revjava design is quite generic and the implementation operates on compiled Java class files. RevJava acts as an assistant to Java coders via critics that can identify potential design and style improvements of the Java code. Figure 13 shows the interface of RevJava critics. RevJava comprises: a model reader, repository, meta-model, property definitions, critic definitions, property evaluator, metrics database, reporting and visualisation.

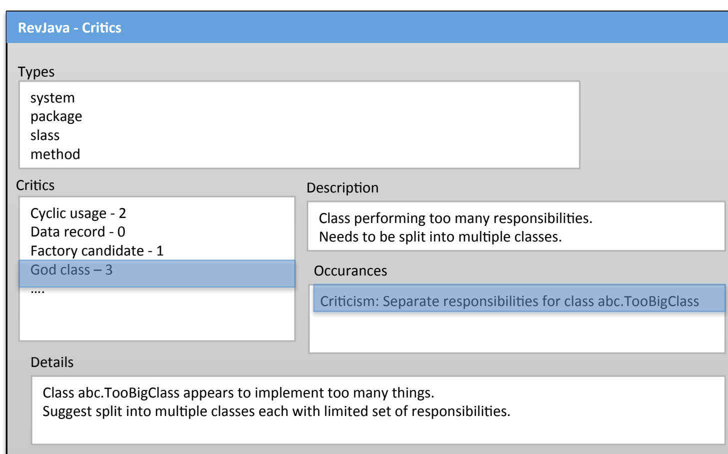

Figure 13. Mockup of the RevJava Critics interface in use.

The model reader reads in the Java code and saves it in a repository, arranged based on a meta-model that identifies all relevant entities in an OO/Java program. For each meta-model type, information about a model element (property and critic) is defined and derived. The property and critic definition is then loaded into RevJava and can be obtained on request. For example, when a user loads a program, some of the properties are treated as “critics” and “metrics”. The information collected via critics and metrics can then be manipulated in different kinds of reporting and visualisations

tools. Visualisations that highlight specific violations in large collections of classes are able to be produced. In addition, RevJava allows Java users to enable and disable critics via a menu. Details can be found at this link: http://www.serc.nl/people/florijn/work/designchecking/RevJava.htm. RevJava and the four previous tools are similar as they all support software development activities. However, the nature of the critiques varies. While the other four tools support design critics based on the design domain, the RevJava tool is a conformance-checking tool for Java programs that helps software developers to detect and correct potential coding mistakes (code domain). Figure 14 shows the mapping of the RevJava tool to the critic taxonomy. Note the differences of this critic tool with respect to the preceding four critic tool characterisations. In particular, RevJava supports minimal rule authoring and provides explanation and suggestion-based feedback. The critic realisation approach is programming code only.

			Critic Domain:	Software Engineering	(Java code)	
Critiquing Approach	**Modes of Critic Feedback**	**Critic Rule Authoring**	**Critic Realisation Approach**	**Critic Dimension**	**Types of Critic Feedback**	**Types of Critic**
C	**T**	IR	RB	**Ave**	**E**	**Co**
A	**G&3D**	MR	KB	**Pve**	**Ar**	**Com**
	MM	DR	Pr	Rve	**Sug**	Con
		E/DR	PM	Pro	Ex	Opt
		AR	OCL	**Lo**	Sim	Alt
			PC	Gl	Dem	Evo
					Int	Pre
					PosF	Tool
					NegF	Exp
					ConF	Org
						Pat
						Str
						Nam
						Met

Figure 14: The mapping of the RevJava tool to the critic taxonomy.

4. 7. Java Critiquer

Qiu and Riesbeck [53-54] describe Java Critiquer, an educational critic tool for Java programming. Java Critiquer's critics are developed using an incremental authoring approach [53-54]). The tool supports both teachers and students. Teachers use the Java Critiquer to critique student java code. Student java code is pasted into a textbox and the Java Critiquer performs automatic critiquing via a pattern matching approach. When a pattern is matched, its corresponding critique is inserted below the problematic Java source code. The teacher then validates the critiques, modifying or removing inappropriate ones as needed. After reviewing the critiques generated by the tool, the teacher can perform manual critiquing. This complements the automatic critiquing to ensure the quality of tool critiquing in the early development stage. Java Critquer allows teachers to add new critiques or reuse existing critiques stored in a database of reusable critiques [53-54]. Figure 15 shows part of the Java Critiquer interface while the tool is in use [53].

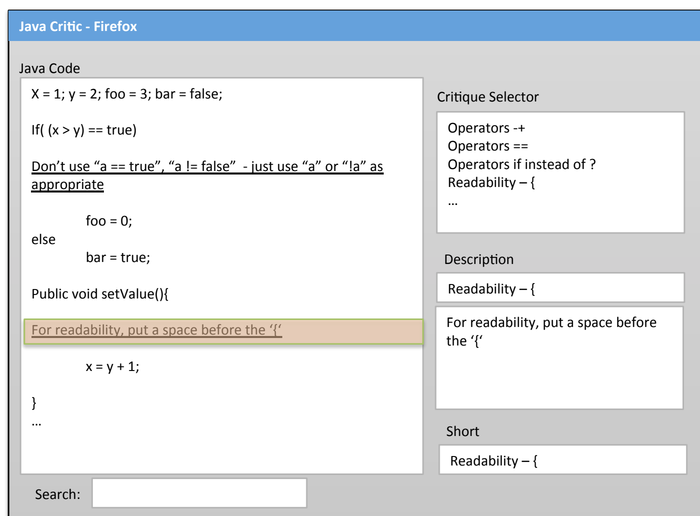

Figure 15. Mockup of the Java Critiquer interface in use.

Java Critiquer is effective as it supports both teachers and students. It helps teachers perform automatic program critiquing reducing their work reviewing student java code. Students can obtain support from Java Critiquer via feedback prior to submitting their assignments and can also use it to undertake self-learning. Java Critiquer is similar to RevJava as both

are critiquing tools for the Java-code domain, and differ from the other four tools, which focus on design. Although Java Critiquer and RevJava provide Java-code critics, the critiquing environment is different. Java Critiquer is developed for a teaching and learning domain, which help teachers and students in a Java course program. RevJava, along with DAISY, IDEA, ABCDE-Critic and ArgoUML are developed for the software development domain to support software professionals in their software development tasks. Figure 16 shows the mapping of the Java Critiquer tool to the critic taxonomy.

			Critic Domain:	Education	(Java coding)	
Critiquing Approach	**Modes of Critic Feedback**	**Critic Rule Authoring**	**Critic Realisation Approach**	**Critic Dimension**	**Types of Critic Feedback**	**Types of Critic**
C	**T**	**IR**	RB	Ave	**E**	**Co**
A	G&3D	**MR**	KB	**Pve**	Ar	**Com**
	MM	**DR**	Pr	Rve	**Sug**	Con
		E/DR	**PM**	Pro	Ex	Opt
		AR	OCL	**Lo**	Sim	Alt
			PC	Gl	Dem	Evo
					Int	Pre
					PosF	Tool
					NegF	Exp
					ConF	Org
						Pat
						Str
						Nam
						Met

Figure 16. The mapping of the Java Critiquer tool to the critic taxonomy.

4. 8. Design Evaluator

The Design Evaluator is a pen-based critic system for design sketching [48]. Its aim is to assist designers, who draw and then justify their drawings, to resolve design problems. Oh et al. [48] demonstrated the sketch based critic system with two applications: 1) architectural floor plan, and 2) web page layout. The Design Evaluator has two components: the first allows the system to access knowledge about domains and the second allows the system to present critic feedback. The Design Evaluator supports designers with critical effective feedback and reasoning about their design sketches. Feedback is in the form of criticism and advice. The way the Design Evaluator presents the critic feedback is very rich as critiques are displayed in various formats: textual, graphical annotation, 3D annotated walk-through models (e.g. architectural floor plans -Figure 17 (left)) and a case library (e.g. web page layout-Figure 17 (right)) [48]. Its use of more than one format to communicate information about the design makes it more understandable to users.

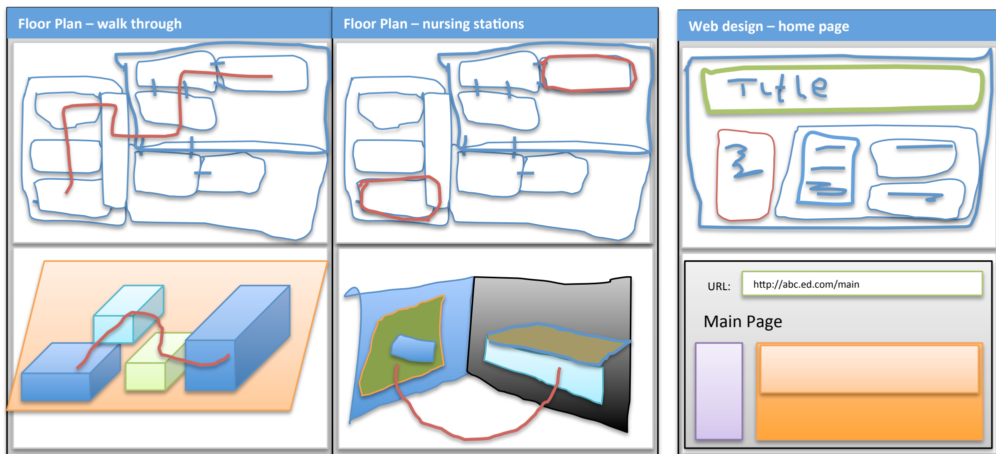

Figure 17. Mockups of critique visualizations in Design Evaluator: architectural floor plan (left) and web page layout (right).

The Design Evaluator is composed of three layers: description, evaluation and visualization. These support different design activities. The *description layer* captures sketching data from the designer and applies some pre-processing to generate a design representation. This is then used by the evaluation layer, which comprises rules coded as Lisp predicates that apply to the design objects. Rules are stored in a list that the designer can check and edit. Each rule expression is associated with a text critic, as well as code that specifies how to annotate the sketch when the critic is applied. A rule may carry additional information used by auxiliary visualization routines such as a VRML model creator (for architecture) or the URL of a representative example case (for web page layout design evaluation). The *visualization layer* then

presents critiques (critic feedback) in textual and visual form. A positive aspect of Design Evaluator is that it provides the ability to link critiques directly onto the design sketch. This is very useful as it retains the designer's focus on the sketches s/he is making. The Design Evaluator is developed for a sketch drawing domain which is completely different from the previous critiquing tools. The main element that distinguishes Design Evaluator from them is the 3D visualisation critic feedback. The Java Critiquer and Submit! tools offer textual critic feedback and the other five tools provide only textual and visual (2D) critic feedback. In addition, Design Evaluator can display generated critiques via a colour coded 3D visualisation. Figure 18 shows the mapping of the Design Evaluator tool to the critic taxonomy.

			Critic Domain:	**Design Engineering**	**(design sketching)**	
Critiquing Approach	**Modes of Critic Feedback**	**Critic Rule Authoring**	**Critic Realisation Approach**	**Critic Dimension**	**Types of Critic Feedback**	**Types of Critic**
C	**T**	**IR**	RB	**Ave**	**E**	**Co**
A	**G&3D**	**MR**	**KB**	Pve	Ar	**Com**
	MM	**DR**	**Pr**	Rve	**Sug**	Con
		E/DR	PM	Pro	**Ex**	Opt
		AR	OCL	**Lo**	Sim	Alt
			PC	Gl	Dem	Evo
					Int	Pre
					PosF	Tool
					NegF	Exp
					ConF	Org
						Pat
						Str
						Nam
						Met

Figure 18. The mapping of the Design Evaluator tool to the critic taxonomy.

4. 9. ClassCompass

ClassCompass is an educational critic tool for software design: "an automated software design critique system with critics that comment on high-level design issues rather than diagram completeness" [19]. It is a collaborative software design tool aimed to assist students as well as instructors in software design activities. Students use the system to produce software designs based on a set of requirements. They can obtain automated feedback (critiques) regarding typical design problems while they perform design tasks. ClassCompass also allows students to manually critique other student's design task. Thus, students can see and learn design styles from the critiques generated by the system as well as from other students. Instructors use an extended version that provides additional features for managing instructional sessions. They use a Web application to configure ClassCompass before students take part in the collaborative design tasks specifying the design principles to be used by students to evaluate designs manually. The instructor uses the ClassCompass client to automatically exchange designs between groups of students.

ClassCompass supports automated and manual critiquing. The automated critiquing executes when a user starts creating a design model. When a critic finds a potential design flaw, an entry is added to a list of critiques beside the design diagram. The critics are not intrusive, as the user can continue their task if they decide to ignore the automated critiques. Users can select an item of interest in the critiques box. A detailed explanation of that particular critique is then presented. The textual critique details are arranged in three parts: 1) *Critique*-describes the design error, 2) *Rationale*-explains why the identified error can reduce software quality, and 3) *Suggestion*-provides a suggestion to correct the identified error. ClassCompass can also highlight the relevant part of the design diagram structure to focus user's attention on the detected problem. Critics in ClassCompass are implemented in Java as pluggable classes that check for a particular pattern in an object model representing the design. Figure 19 shows the user interface of ClassCompass with automated critiquing.

ClassCompass is similar to Java Critiquer and Submit! as these critiquing tools are developed for the education domain. However, the education focus of these tools is different. ClassCompass focuses on software design education, while Java Critiquer and Submit! focus on computer programming education. These critiquing tools aim to support teachers (or instructors) and students by offering advice and guidance related to their particular education. This is in contrast to earlier surveyed tools, such as ArgoUML, ABCDE-Critic, DAISY and IDEA that are aimed at software professionals. ClassCompass has similarities to these four tools as they all provide design critics to resolve potential design mistakes. However, design critics offered by ClassCompass focus on higher-level design issues [19]. Furthermore, ClassCompass supports automated and collaborative support for software design education. The feature of collaboration with distributed users is not offered by most of the preceding tools mentioned above. Figure 20 shows the mapping of the ClassCompass tool to the critic taxonomy.

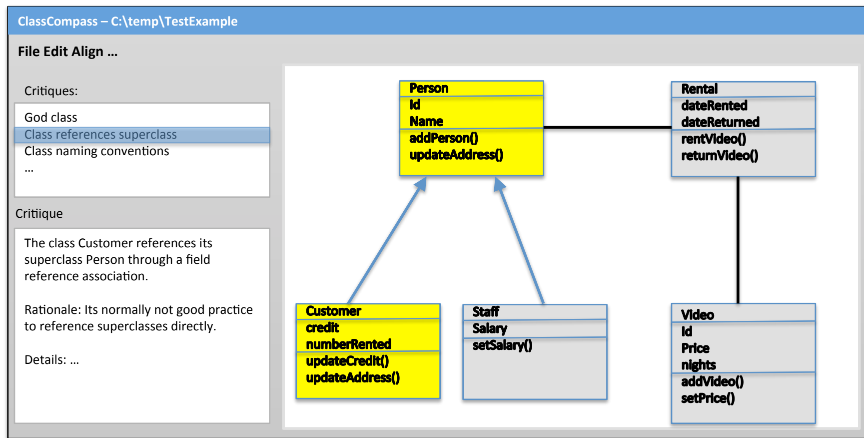

Figure 19. Mockup of the ClassCompass user interface in use.

			Critic Domain:	Education	(UML class and sequence diagrams)	
Critiquing Approach	Modes of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Dimension	Types of Critic Feedback	Types of Critic
C	**T**	IR	RB	**Ave**	**E**	**Co**
A	**G&3D**	MR	KB	Pve	**Ar**	Com
	MM	DR	Pr	Rve	**Sug**	Con
		E/DR	**PM**	Pro	Ex	Opt
		AR	OCL	**Lo**	Sim	Alt
			PC	Gl	Dem	Evo
					Int	Pre
					PosF	Tool
					NegF	Exp
					ConF	Org
						Pat
						Str
						Nam
						Met

Figure 20. The mapping of the ClassCompass tool to the critic taxonomy.

4. 10. ECS

Essay critiquing system (ECS) is a web-based system that offers immediate feedback on writing ideas to students on an essay topic by comparing student and model essays [41]. ECS uses latent semantic analysis (LSA), an automatic text analysis technique for computing the semantic similarity between pieces of textual information, with the support of a large corpus of students' essays. The components of ECS are: teacher input, student input, a database to store student answers and reference materials from external sources, a text segmentation and a pre-processing engine, an LSA engine, and a semantic matcher for providing critical feedback to students. ECS offers two modes of feedback to the students: 1) feedback on content and sub-themes of the essay, and 2) organisation of arguments in the essay.

When a student submits an essay to ECS for feedback, the semantic similarity between all possible pairs (i.e. one from the student's essay and the other from the system's sub-theme list) is computed. Any missing sub-themes from the student's essay will be detected and reported to the student for his/her consideration when revising the essay. In addition, the system is able to recognize text segments in the student's essay that match with the sub-themes in the sub-theme list, and highlight them in different colours according to the degree of matching [41].

The process of critiquing essays and generating effective feedback is not an easy task. For instance, the process of handling essay critiquing and program critiquing might be different. Program critiquing, as used in RevJava and Java-Critiquer, is feasible because the programming language has a well-defined syntax and semantics that makes the task of critiquing easier than it would be for submissions such as essays [51]. Although the study by Lee et al., shown that ECS is a useful tool to facilitate students' writing, more work on the tool should be done, specifically on the corpus development which provide more essay topics for students' reference [41]. Figure 21 shows a mapping of ECS to our critic taxonomy. Cheung, one of the researchers of ECS, has confirmed the mapping of ECS to our critic taxonomy [18].

			Critic Domain:	Education	(Essay critic)	
Critiquing Approach	**Modes of Critic Feedback**	**Critic Rule Authoring**	**Critic Realisation Approach**	**Critic Dimension**	**Types of Critic Feedback**	**Types of Critic**
C	**T**	IR	RB	**Ave**	**E**	**Co**
A	G&3D	MR	KB	**Pve**	**Ar**	**Com**
	MM	DR	Pr	Rve	**Sug**	Con
		E/DR	**PM**	Pro	Ex	Opt
		AR	OCL	**Lo**	Sim	Alt
			PC	Gl	Dem	Evo
					Int	Pre
					PosF	Tool
					NegF	Exp
					ConF	Org
						Pat
						Str
						Nam
						Met

Figure 21. The mapping of the Essay Critiquing System (ECS) to the critic taxonomy.

4. 11. FFDC

Flat-pack Furniture Design Critic (FFDC) supports design learning in studio settings [50]. It uses a constraint based design critic approach that provides students feedback with five delivery types (interpretation, introduction, example, demonstration and evaluation) and three communication modes (written comments, graphical annotations, and images). FFDC selects specific methods to present feedback by considering a specific user's knowledge and the critiquing methods that the program has previously used for the user. Figure 22 shows an example of the style of graphical and 3D visualisation feedback used in FFDC.

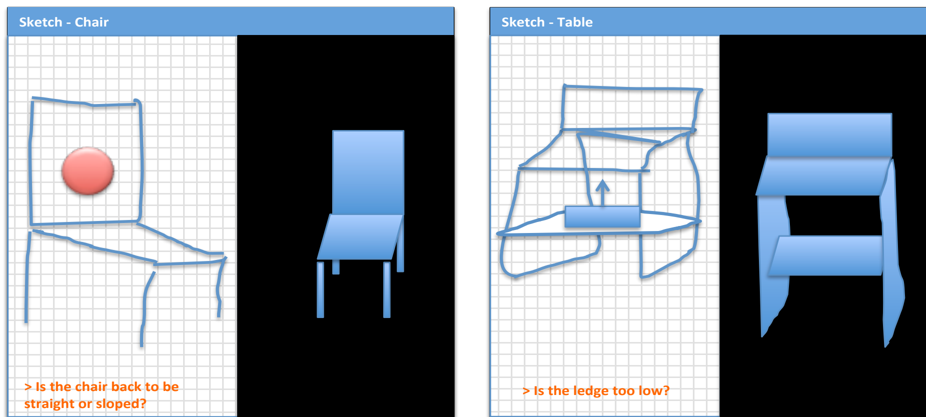

Figure 22. Mockup of Flat-pack Furniture Design Critics in use.

FFDC is written in Macintosh Common Lisp using OpenGL to provide 3D models and the Lisa (Lisp-based Intelligent Software Agent) production rule system to justify a planned furniture design using stored constraints. It has eight components: *Construction Interface, Parser, Pattern Matcher, Design Constraints, Critiquing Rules, User Model, Pedagogical Module,* and *Critiquer*. The *construction interface* allows a user to perform design sketching via a stylus and digitising tablet. All sketched glyphs are recorded, a Cartesian coordinate system is defined and a 3D model generated. The *parser* parses the sketched diagram and 3D model to generate two kinds of data: 1) parts and their properties and 2) part configurations. A symbolic representation of the designed furniture item is then saved. A set of *design constraints* represents the principles that designers need to know to design furniture. Twenty-seven structural constraints and 36 functional constraints are implemented. The *pattern matcher* compares the symbolic representation of the design against the design constraints to detect critic violations. The *user model* component has short-term and long-term user models. The former stores the results of the *pattern matcher* and the latter saves a history of all violated and satisfied constraints over multiple critiquing sessions. The *pedagogical module* takes input from the short- and long-term user models and decides specific critiquing methods to use via the *critiquing rules*. These determine which delivery types and communication modes are to be used in a specific case. For instance, if a designer is recognised as a novice, the *pedagogical module* will choose 'demonstration' delivery type rather than 'example' as novices normally have trouble using examples in their designs. The *critiquer* presents the critique to the designer. It comprises three modules: 1) *text critique* - presents written comments, 2) *example finder* - selects relevant examples, and 3) *graphic critique* - highlights relevant furniture parts and

draws graphical annotations. The tool offers feedback (critiques) in several ways based on users' knowledge and previously used feedback.

Although FFDC tool is similar to the Submit!, Java Critiquer, ClassCompass and ECS tools due to education domain application, the critiques focus is different. FFDC critics produce critiques to help instructors and students in furniture design education. This is in contrast to the Submit!, Java Critiquer, ClassCompass and ECS that generate critiques for software programming, software design and essay writing education. The FFDC is similar to the Design Evaluator as these critiquing tools provide textual, graphical and 3D visualisation feedback. In addition, the FFDC offers five types of critic feedback: interpretation, introduction, example, demonstration and evaluation (positive and negative). This is in contrast to earlier surveyed tools, such as ArgoUML, ABCDE-Critic, DAISY, IDEA, RevJava, Java Critiquer, Class Compass and ECS that provide at most two to three types of critic feedback. FFDC does not support critic authoring. Figure 23 shows the mapping of the FFDC tool to our critic taxonomy.

			Critic Domain:	Education	(furniture design)	
Critiquing Approach	**Modes of Critic Feedback**	**Critic Rule Authoring**	**Critic Realisation Approach**	**Critic Dimension**	**Types of Critic Feedback**	**Types of Critic**
C	**T**	IR	**RB**	**Ave**	**E**	**Co**
A	**G&3D**	MR	KB	Pve	Ar	Com
	MM	DR	Pr	Rve	**Sug**	**Con**
		E/DR	**PM**	**Pro**	**Ex**	Opt
		AR	OCL	**Lo**	Sim	Alt
			PC	Gl	**Dem**	Evo
					Int	Pre
					PosF	Tool
					NegF	Exp
					ConF	Org
						Pat
						Str
						Nam
						Met

Figure 23. The mapping of the FFDC tool to the critic taxonomy.

4. 12. HeRA

Heuristic Requirements Assistant (HeRA) is a feedback centric requirements editor supporting analysts with information based on several heuristic feedback facilities [39]. Requirements analysts receive warnings and hints on detection of ambiguities or incomplete requirements specifications while typing/writing requirements.

HeRA comprises three editors and two components: 1) general purpose requirements editor, 2) use case editor, 3) glossary editor, 4) argumentation component and 5) simulation component. Requirements are constructed using the three editors and produce domain specific artefacts i.e. requirements, use cases and a glossary. HeRA allows users to argue with the critiques via the argumentation component. This can help users clarify their understanding of requirements problems and leads to improved heuristics feedback in future. The simulation component provides 'what-if' analysis and derives three models as the use case is written: *UML Use Case Diagrams*, *EPC Business Processes*, and *Use Case Points Estimations*. These models can provide extra information (feedback) to the requirements author regarding the requirements being documented [39]. Heuristics rules are defined in JavaScript and can access the data model of the requirements editor. HeRA also provides wizards to assist requirements authors to generate Java script code for a rule. Rules can be modified and applied directly. Thus, new critiques are shown immediately. Users have the option to fix or ignore critiques offered to them. In general, HeRA offers different levels of feedback to the analyst via the argumentation and simulation components.

HeRA is comparable to the earlier critiquing tools that aim to assist software professionals by offering guidance via critiques. However, HeRA critics differ from these critiquing tools because HeRA produces critiques that focus on more abstract software requirement tasks. HeRA also generates critiques via an argument style, which is similar to ABCDE-Critic. HeRA supports a range of critic authoring mechanisms but these are implemented solely in code. Figure 24 shows the mapping of the HeRA tool to the critic taxonomy. Knauss, one of the developers of HeRA, has confirmed the mapping of HeRA to our critic taxonomy [40].

			Critic Domain:	Software Engineering	(requirements engineering)	
Critiquing Approach	Modes of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Dimension	Types of Critic Feedback	Types of Critic
C	**T**	**IR**	RB	Ave	**E**	Co
A	G&3D	**MR**	KB	Pve	**Ar**	**Com**
	MM	**DR**	Pr	Rve	Sug	**Con**
		E/DR	PM	**Pro**	Ex	Opt
		AR	OCL	**Lo**	**Sim**	Alt
			PC	**Gl**	Dem	Evo
					Int	Pre
					PosF	Tool
					NegF	Exp
					ConF	Org
						Pat
						Str
						Nam
						Met

Figure 24. The mapping of the HeRA tool to the critic taxonomy.

4. 13. Marama Critic Definer

Inspired by the existing critic tools work, we have applied similar ideas to our Marama meta-modelling toolset [7, 8, 35]. Marama is implemented as set of Eclipse-plugins and includes meta-tools as well as modelling tools. Our meta-tools are used to generate complex visual modelling tools, and these modelling tools could benefit from the addition of various critics. Thus, we wanted to extend our Marama meta-tools by embedding a generic critic specification component to assist end-user tool developers to specify and generate critics efficiently and easily for domain-specific visual language (DSVL) tools. Our critic authoring tool, *Marama Critic Definer* uses a visual and template-based approach to support the task of end user specification of critics and feedback using examples for Marama-based DSVL tools [7]. End-user tool developers can specify critics for their DSVL tools without the need for in-depth technical knowledge of critic construction. We also provide a critic authoring template-based approach as an alternate style for the critic specification task. End-user tool developers can customise critics and introduce new critic templates via a critic authoring guideline and critic template editor. Our approach is based on using information expressed in a meta-diagram (the Marama meta-model diagram) as input for critics to be realized in a model diagram (i.e. a diagram constructed using the realized modelling tool specified by the meta-model). The Marama meta-model diagram is expressed using an Extended Entity Relationship (EER) notation, but it is important to note that our approach is only minimally dependent on this notation. If a richer notation were available, more information could be extracted and used for specifying critics and checking user diagrams. A description is shown in Figure 25.

The pre-existing Marama meta-tool set has three key DSVL tool specification editors: the meta-model definer view to define a tool's information model; the shape designer view to define the visual notation elements; and the viewtype definer view to specify the mappings of meta-elements to visual representations. Once a new tool is defined and equipped with sufficient information an end-user tool developer can then select the new *Marama critic definer* view to author and realize critics for their target DSVL tool specification. A new Marama-based tool with critic support is then generated as a set of plug-ins. A tool end-user can then create new modelling projects and diagrams using the new tool. When a diagram is created, critics for that particular tool are instantiated. If a user creates design content that a critic identifies as problematic, a critique is generated to notify the user about the potential problems/errors. This is shown in Figure 26. Feedback from the critic is displayed to allow the user to fix the problem/error.

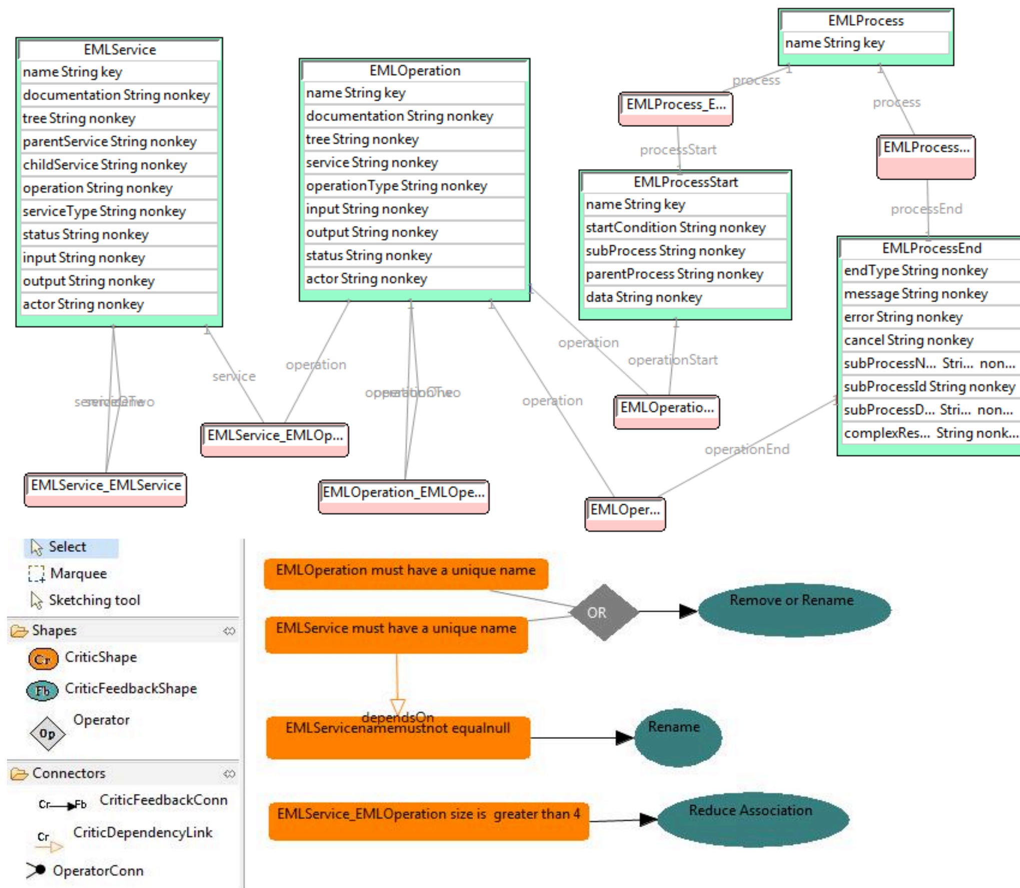

Figure 25. Critics specified in the critic definer editor (bottom) based on the meta-model of a simplified MaramaEML tool defined in the meta-model editor (top).

The Marama critic definer editor is different from the earlier surveyed tools as it is embedded within a meta-modelling tool that implements DSVL tools. The twelve preceding critiquing tools are not meta-tools but have a fixed-type of application domain. For instance, ArgoUML was developed for the domains of UML (Unified Modelling Language) and HeRA was built for the software requirements domain. Likewise, Java Critiquer and RevJava were developed for the domain of Java programming. The Marama meta-tool is used to generate visual modelling tools, and these modelling tools could benefit from the addition of various critics. Thus, the Marama critic definer editor can specifically assist end-user tool developers to specify critics for DSVL tools. For instance, a simplified MaramaEML modelling tool developed using the Marama meta-tool is shown in Figure 26 and once critics are specified by the tool developer critiques can be generated (as shown in Figure 27). Therefore, any domain of modelling tools can be developed in the Marama-meta tool and critics can be specified based on the modelling tool domains. The earlier surveyed tools lack this feature.

The Marama critic definer editor provides a visual way of expressing and specifying critics for DSVL tools. Notational representation of a critic authoring capability is offered to end-user tool developers to specify critics for their DSVL tools. In addition, a template-based approach, which was designed in a form-based interface, allows easier input of critic templates into a DSVL tool environment. The combination of a notational representation and a critic-authoring template-based approach is another useful approach to support end-user tool developers in the critic specification task. This is in contrast with the twelve previous tools where critics are not specified visually and where the capability of specifying critics is the responsibility of skilled and knowledgeable software developers. Figure 28 shows the mapping of the Marama critic definer tool to the critic taxonomy.

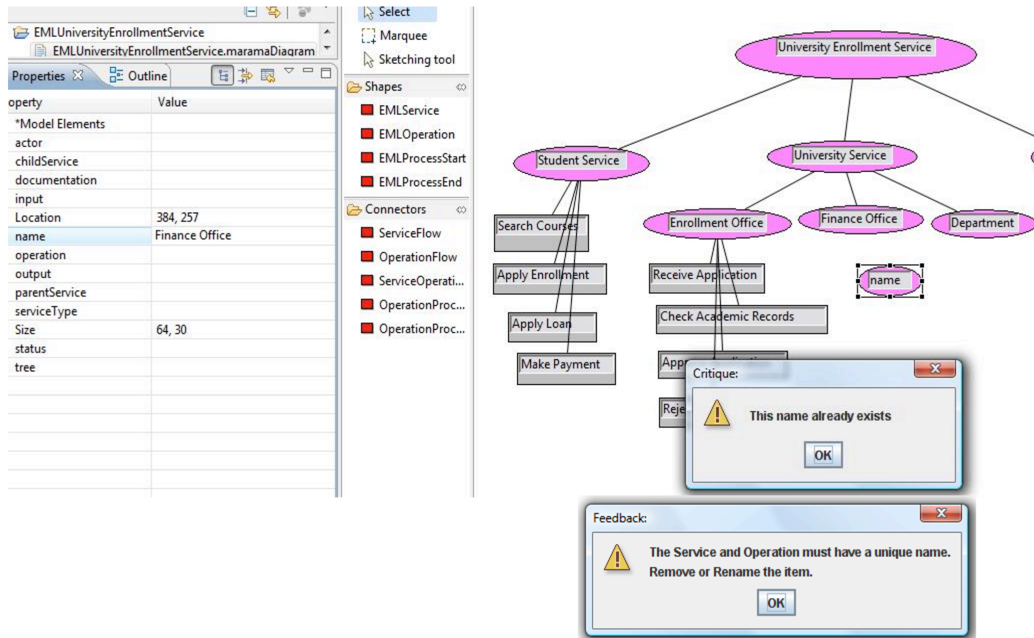

Figure 26. A critique message is displayed when a uniqueness name critic is violated.

			Critic Domain:	Software Engineering	(DSVL tools)	
Critiquing Approach	**Modes of Critic Feedback**	**Critic Rule Authoring**	**Critic Realisation Approach**	**Critic Dimension**	**Types of Critic Feedback**	**Types of Critic**
C	T	IR	RB	Ave	E	Co
A	G&3D	MR	KB	Pve	Ar	Com
	MM	DR	Pr	Rve	Sug	Con
		E/DR	PM	Pro	Ex	Opt
		AR	OCL	Lo	Sim	Alt
			PC	Gl	Dem	Evo
					Int	Pre
					PosF	Tool
					NegF	Exp
					ConF	Org
						Pat
						Str
						Nam
						Met

Figure 27. The mapping of the Marama critic definer tool to the critic taxonomy.

5. Comparison and Discussion

We have proposed and illustrated a critic taxonomy based on aspects that characterize critics (or critiquing systems). These aspects were gathered from the broad critic literature. Our critic taxonomy identifies eight characteristic groups: critic domain, critiquing approach, modes of critic feedback, critic rule authoring, critic realization approach, critic strategy, types of critic feedback, and types of critic.

To provide a higher-level view of coverage of the various taxonomic elements, Figure 28 shows a simple heatmap visualisation where depth of colour represents numbers of tools exhibiting a particular characteristic from the 13 exemplar tools categorised using our taxonomy above. Data with low and no value displayed in the heatmap indicate areas where potential research/future work might be undertaken in exploring seldom-used critic approaches. Alternatively, it may be that these seldom-used approaches have very narrow applicability compared to others.

Critic Domain:						
Critiquing Approach	Mode of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Strategy	Type of Critic Feedback	Type of Critic
C: 3	T: 13	IR: 9	RB: 4	Ave: 9	E: 11	Co: 10
A: 10	G&3D: 9	MR: 8	KB: 2	Pve: 7	Ar: 5	Com: 9
	MM: 0	DR: 7	Pr: 2	Rve: 0	Sug: 12	Con: 5
		E/DR: 9	PM: 6	Pro: 3	Ex: 2	Opt: 1
		AR: 7	OCL: 0	Lo: 13	Sim: 1	Alt: 1
			PC: 11	Gl: 2	Dem: 1	Evo: 1
					Int: 1	Pre: 1
					PosF: 3	Tool: 1
					NegF: 2	Exp: 1
					ConF: 1	Org: 1
						Pat: 2
						Str: 3
						Nam: 2
						Met: 2

Figure 28. Heatmap showing frequency of occurrence of different characteristics in the 13 exemplar tools.

Critic domain is the first group defined in the critic taxonomy. Seven tools from the software engineering (SE) domain, five tools from the education domain and one tool from the design engineering/architecture domain were examined. Our analysis of these tools shows that critics can be applied in a wide variety of domains but are specified based on the domain knowledge of that particular area. This is aligned to the argument made by Fischer et al. that critics are most effective when they are embedded in domain-oriented design environments [28]. We identified the critic domain in our critic taxonomy as being crucial as it appears to greatly influence critic development. The other groups defined in our critic taxonomy are dependent on the critic domain as the type of domain knowledge they present influences the choice of elements in each other group. We speculate that there is a strong dependency between the critic domain and the other groups in our critic taxonomy and discuss this below.

The second taxonomy group is the *critiquing approach*. All critic tools apply some kind of critiquing approach to allow reasoning to detect potential problems in the user's work or design solution. Our finding from the13 exemplar tools categorised is that most tools use an analytical approach. Those categorised tools employing comparative critiquing – ClassCompass, ECS and FFDC – are in the domain of education. They adopt comparative critiquing, as opposed to analytical, since they store a set of software design principles (for ClassCompass), model essays (for ECS) and design constraints (for FFDC) to be compared against users' design/solution. In addition, this enables guiding of learners to solutions.

The following taxonomy group is the *mode of critic feedback* that concerns the format of feedback to be displayed for users. The thirteen exemplar tools use a textual critic feedback approach as the default mode of critic feedback. Four tools only provide textual format to present critiques to their users: Submit!, JavaCritiquer, ECS and HeRA. Nine other tools have augmented their critiques modes via graphical and 3D visualisation. The Design Evaluator and FFDC are the only tools that offer 3D visualisation with their graphical and textual critiques. The other seven tools: ArgoUML, ABCDE-Critic, IDEA, RevJava, DAISY, ClassCompass and MaramaCritic present their critiques via textual and graphical mode. Though none of the thirteen tools apply the 'multi-modal element' it can be considered for future use. Furthermore, it is an advantage to offer critiques in various modes.

Critic rule authoring is the fourth taxonomy group. Six of the thirteen exemplar tools provide all five functions listed in this group: ABCDE-Critic, IDEA, DAISY, Java Critiquer, HeRA and Marama Critic designer. ArgoUML allows an end user to enable/disable rules. However, any new rules have to be added using Java by a tool developer. A function to delete rules is not offered to users. The Design Evaluator allows the end user (designer) to inspect and edit rules, but a function to enable and disable critics is not provided. In contrast, RevJava allows users to enable and disable critics via a menu option. However RevJava does not provide the ability for users to author or edit critic rules. Three tools that do not provide any such facility are ClassCompass, ECS and FFDC. Here, the system/tool designers write critic rules in advance and facilities to customize rules are not provided to users. Providing users with a facility to author or customize critic rules is a useful element to be considered. Realising that critiquing capacity and issues may change from time to time, it is worth allowing users to author or customise (add, delete, modify) their own critic rules for a particular critic domain. However, it is possible for end users in particular to "break" the critics by specifying bad rules or disabling or modifying important ones. The Marama critic definer tool enables tool end users, as well as tool designers, to perform all the functions defined in our taxonomy. Unlike the other critic systems surveyed, Marama critic designer was built from scratch to support this range of functionality for both end users and tool developers. However, we have found that some kinds of critic tools, critic rule approaches, tool users and domains are more amenable to this approach than others [2]. In

addition, Fischer and Girgensohn [26] provide a framework for end-user modifiability that allows users to modify the design environment itself [26]. According to them, end-user modifiability is a requirement in cases where the systems do not match to a particular task, working style or personal sense of aesthetics [26]. However, several issues and challenges need to be considered when implementing end user modifiability. One of the issues concerns supporting changes [26] where the users must have a clear understanding of the implications of the modification with respect to the problem domain [26]. Other issues are explained in [26]. The decision to allow users to author or customise a given critic can therefore be framed as an investment equation, in which the expected payoff is compared to the investment and risk [11]. Thus, we could apply the Attention Investment model [11] for the critic authoring issue to include the cost-benefit analysis. However, we have currently left this as future work.

The next group in our critic taxonomy is the *critic realisation approach,* which details how critics are implemented or specified in a system/tool. In the thirteen exemplar tools, most implement critics via programming code, sometimes with the help of design patterns and class hierarchies to ease critic programming. Rule-based and pattern-matching are other widely used approaches that do not require programming per se. Most systems/tools apply more than one approach. OCL is widely used in meta-modelling tools to specify tool constraints and can be used to specify critics as reported by Bezivin and Jouault [10]. However, our early attempts to use an OCL approach in our Marama Critic Definer showed this to provide a strong barrier to use for end-user tool developers [8]. Many critic realisation approaches are too difficult for tool end users to specify critics to any great level of detail due to reliance on programming, complex constraints or complex rule-based systems. The critics' realisation approach facilitates tool developers in identifying and using an appropriate approach to realise their critics. Each approach has pros and cons that a critic developer needs to take into account. How critics are implemented closely relates to the both the critiquing approach used in the system and who is going to maintain the critics. Thus carefully considering the critiquing approach is also a useful dimension to assist a critic developer in deciding whether comparative, analytical or both approaches are to be used in critic development.

Critic strategy is a key aspect that critic developers need to consider when adopting critics in their system/tool as this determines how tool end users perceive the "help" (or hindrance) of the critics in a tool. Most of the thirteen exemplar tools apply both active and passive critics. The Design Evaluator only provides active critics when a design sketching activity triggers a critic. Java Critiquer prefers to use passive critics. This passive critic choice was made as the tool developers wanted students to learn from mistakes when they code their Java programs. Three tools provide proactive critics (i.e. critics that make an attempt to make changes to the diagram) to their users: ABCDE-Critic, IDEA and HeRA. All of the categorised tools provide local critics but only two tools (DAISY and HeRA) also offer global critics that consider a global perspective or interaction with other elements in a design/solution. A critic either provides intervention strategies, activation strategies or timing strategies. Most of the articles we reviewed do not discuss explicitly the choice of the critic strategies in developing a critic tool. However, the results of mapping the thirteen tools with this element suggested that there is a range of critic strategies that can be used to enhance/improve critic development.

The seventh group in our critic taxonomy is the *type of critic feedback,* which refers to the techniques used to present critic feedback (or critiques) to users. A variety of techniques are used in the thirteen exemplar tools. The most common, used in all tools, are explanations, suggestions and argumentations. However, a few tools add richness to critic feedback in the form of constructive feedback, positive and negative feedback, examples, interpretations, simulation and demonstration. Tools that provide multiple critic feedbacks to users are: ArgoUML, Design Evaluator, FFDC, and HeRA. We believe that multiple critic feedbacks presented by these tools aimed to assist users develop their solutions or design tasks better and to improve users' learning. Conventional critics normally provide only a textual critique but tool developers should recognise the benefit of combining several modes in presenting critiques.

The last group in our critic taxonomy is the *type of critic* offered by a system/tool to its users. Most of the thirteen exemplar tools offer correctness and completeness critics. We found from our categorisation of these exemplar tools that the types of critics chosen depend heavily on the critic domain defined by a system/tool. Targeted end users of a particular critic tool can be an influence factor too in determining critic types. For that reason, appropriate and relevant critics need to be designed by the critic developer. Critic types from the taxonomy are also helpful in guiding the types of critics that can be considered by a critic developer other than the common critic types i.e. correctness and completeness critics. Critic developers are encouraged to consider incorporating other types of critics in a system. For instance, the ClassCompass presents structure, naming and metric critics. The DAISY, FFDC and HeRA tools offer consistency critics. Likewise IDEA provides design pattern critics.

Finally, though we selected thirteen exemplar tools mostly from the software engineering domain (including our own critic design and realisation tool) to illustrate our taxonomy application example, the taxonomy is applicable to critics in other domains. We have shown this through characterising the Design Evaluator, FFDC, ECS, ClassCompass, Submit! and Java Critiquer tools. These support education in a variety of domains including floor planning, design sketches, essays writing, software design, and computer programming (such as Java programming). Our contribution is to present a comprehensive taxonomy of critics by carrying out an analysis of existing critic approaches that allows critic developers and critic tool end users to better group tools, techniques and formalisms based on their common qualities, features,

characteristics and representative elements. This allows tool users and developers to reason about different critic characteristics that could be realised in a critic-supporting tool, sometimes trading off different approaches across the different taxonomy groups.

We have also created additional heat maps illustrating our critic taxonomy application to different critic domains. We selected five critic tools from the group of education-oriented tools and five critic tools from the group of production use tools (from the domain of software engineering). These ten critic tools are from the list of 13 critic tools described in the previous section. We wanted to see whether there are variations between the two types of critic tools (i.e. education-oriented versus the rapid problem solving critic tools). Figure 29 shows the mapping of education-oriented critic tools versus the production-use critic tools to our critic taxonomy.

			Critic Domain:	Five education-oriented tools	(Java Critiquer, ClassCompass, FFDC, Submit!, ECS)	
Critiquing Approach	**Mode of Critic Feedback**	**Critic Rule Authoring**	**Critic Realisation Approach**	**Critic Strategy**	**Type of Critic Feedback**	**Type of Critic**
C: 3	T: 5	IR: 2	RB: 1	Ave: 3	E: 3	Co: 4
A: 2	G&3D: 2	MR: 1	KB: 0	Pve: 2	Ar: 1	Com: 2
	MM: 0	DR: 1	Pr: 0	Rve: 0	Sug: 5	Con: 1
		E/DR: 2	PM: 5	Pro: 0	Ex: 1	Opt: 0
		AR: 1	OCL: 0	Lo: 5	Sim: 0	Alt: 1
			PC: 4	Gl: 0	Dem: 1	Evo: 0
					Int: 1	Pre: 0
					PosF: 2	Tool: 0
					NegF: 1	Exp: 0
					ConF: 0	Org: 0
						Pat: 0
						Str: 2
						Nam: 1
						Met: 1

			Critic Domain:	Five production-use tools (software engineering)	(ArgoUML, ABCDE-Critic, RevJava, HeRA, MaramaCritic)	
Critiquing Approach	**Mode of Critic Feedback**	**Critic Rule Authoring**	**Critic Realisation Approach**	**Critic Strategy**	**Type of Critic Feedback**	**Type of Critic**
C: 0	T: 5	IR: 4	RB: 1	Ave: 4	E: 5	Co: 4
A: 5	G&3D: 4	MR: 4	KB: 0	Pve: 4	Ar: 2	Com: 5
	MM: 0	DR: 3	Pr: 1	Rve: 0	Sug: 5	Con: 3
		E/DR: 5	PM: 1	Pro: 2	Ex: 0	Opt: 1
		AR: 3	OCL: 0	Lo: 5	Sim: 1	Alt: 1
			PC: 5	Gl: 1	Dem: 0	Evo: 1
					Int: 0	Pre: 1
					PosF: 0	Tool: 1
					NegF: 0	Exp: 1
					ConF: 1	Org: 1
						Pat: 2
						Str: 1
						Nam: 1
						Met: 1

Figure 29: Education-oriented versus Production-use critic tools.

A key contrast is that critic developers for production-use tools (i.e. in this case the Software Engineering domain) favour employing an analytical critiquing approach, compared to critic developers for the education-oriented tools that tend towards using a comparative critiquing approach. Our explanation for this is that most emerging design problems in software engineering are ill-structured and must be solved by exploration and error elimination. It is impossible to obtain a complete and correct solution for all kinds of problems in software engineering design activities. In contrast, critic developers for education-oriented tools can use a comparative critiquing approach by providing predefined solutions in the critics thus allowing comparison of a user's solution against the system's "best practice" or correct solution. We believe that the use of comparative critiquing in the education domain aims to support and promote self-directed learning among the students. For instance, ECS compares the student's essay with the model essays which are added to ECS. This provides some evidence that the choice of critiquing approach is influenced by the critic domain.

The use of a textual critic feedback approach seems to be the default mode of critic feedback for both education-oriented tools and production-use tools. Critic developers for the production-use tools (within the Software Engineering domain)

incorporate some graphic annotation with textual feedback critiques in order to provide a better result to their tool users (i.e. software designers and developers). We speculate that the reason for this is because many software engineering design tasks involve diagrammatic representations or visualisations. Thus, providing a graphical solution for the tool users is often helpful. Although critiques can be generated using a multi-modal mode approach such as images, animation, sound, or video, writing such critiques is not easy. Thus, most of the critic feedback in critic tools has predominantly been text-based. Again, we believe that the critic domain affects the choice of critic feedback. If a critic domain does not involve any diagrammatic or model representations for a problem domain, then textual critiques are sufficient to support the tool users' work.

As for critic rule authoring, it seems that the critic developers for production-use tools (software engineering domain) often provide their tool users with capabilities for inserting, deleting and modifying critic rules, enabling or disabling critic rules, and authoring critic rules. However, critic developers for education-oriented critic tools do not tend to offer such kinds of facilities to their tool users. We speculate that using analytical critiquing techniques make it more readily possible for critic developers to provide critic rule authoring facilities (i.e. insert, delete, modify, enable/disable and author critic rules), than compared to using comparative critiquing as used in most of the education-oriented critic tools. Our assumption is that the choice of critiquing approach would influence the critic developers indirectly in providing the critic rule authoring facilities.

Programming code appears to be the most widely used approach in critic realisation for both education-oriented and production-use critic tools. Thus, we can assume that the critic tool developers either prefer or feel they have to use programming code compared to other approaches for realising the critics in both of these problem domain. The pattern matching approach seems to be an easy approach to use in realising the critics for the education-oriented critic tools. We believe that the critic domain might influence the choice of the critic realisation approach. Authoring complex critics requires considerable skill and some critics require considerable flexibility to specify. Lack of suitable frameworks or critic authoring facilities in meta-tools used to build tools with critics may mean tool developers feel they must program critics in code.

Both education-oriented and production-use critic tools adopt a mix of active and passive critics. The types of critic feedback designed by the critic tool developers for the education-oriented and production-use tools are mostly in a form of suggestion and explanation. The types of critic that the critic tool developers work on are largely on correctness, completeness and consistency critics, though any type of critic can theoretically be incorporated in the critic tools. We believe that the critic domain also influences the critic strategy (e.g. active critics, passive critics, etc) types of critic feedback (e.g. explanation, suggestion, etc.) and types of critic (e.g. correctness, completeness, etc). Our argument is that, with a good knowledge of the critic domain, critic developers could choose the most appropriate critic strategy(s), type(s) of critic feedback and type(s) of critic to be offered to the end users.

For example, the development of the Flat-pack Furniture Design Critic (FFDC) [39] needs domain knowledge of design education (specifically in learning furniture design). The choices of textual, graphical and 3-dimension visualisation as the modes of critic feedback are relevant to the tool as they support the students' understanding in learning furniture design. In addition, the nature of design in studio settings has led the FFDC developers to provide several types of critic feedback to students when interacting with the FFDC tool. The choices of types of critic feedback are: explanation, suggestion, examples, demonstration, interpretation, positive feedback, and negative feedback. This would enhance the teaching-learning process of furniture design. The FFDC tool also monitors students' design/work and offers active critics to the students when a potential problem of design is detected. The types of critic offered by the FFDC are mostly on the correctness, consistency and structural critics that are based on the design principles for furniture design.

We have commented above that there appear to be such relationships and speculate that there are trade-offs between choices in the different groups, similar, for example, to the trade-offs that occur between the various dimensions in the Cognitive Dimensions of Notations Framework [25]. We postulate that a better understanding of such dependencies and trade-offs would be of considerable assistance to critic tool developers, but is beyond the scope of this paper and is left for future work. This would allow the taxonomy to be used for design guidance, as the Cognitive Dimensions framework has been used for guidance in notation design.

In closing our discussion of the heatmap analysis, it should be noted that there are numerous issues and challenges that need to be considered in developing critiquing tools. These issues and challenges are described by many researchers such as Fischer et al. [27], Silverman [60-61], Sumner et al. [63], Robins [57] and Irandoust [36]. For instance, Sumner et al. describe the cognitive ergonomic challenge in developing critiquing systems [63]. According to them, "a cognitively ergonomic system should appropriately guide designers' problem solving without hindering their creative processes, and thus positively influence both the design process and the design product" [63]. Other issues and challenges, to name a few are: identifying the best architecture and language for a critic [57, 61], developing a reusable critiquing infrastructure [57] and managing the critic development life-cycle [57]. Recently, Fischer [30] claimed that a key challenge in building

critiquing systems is to exploit context-awareness to say "the 'right' information, at the 'right' time, in the 'right' place, in the 'right' way, to the 'right' person"[30].

The taxonomy proposed here can be considered as defining core characteristics of critic systems. This core taxonomy can be extended to include more specialised characteristics, such as those detailed above, but this is left as future work.

6. Conclusion

We have proposed and illustrated the use of a new critic taxonomy based on a range of aspects that characterize critics (or critiquing systems). These aspects have been gathered from a broad analysis of the critic literature and our attempt to better characterise a wide range of critics. Our critic taxonomy identifies eight characteristic groups: critic domain, critiquing approach, mode of critic feedback, critic rule authoring, critic realization approach, critic dimension, type of critic feedback, and type of critic.

The utility of our critic taxonomy is manifold: to provide an overview of critic research; to identify and distinguish key critic elements; and to recognize techniques or methods applied in critics. We believe that this taxonomy provides a meaningful way of describing and reasoning about critics and a vocabulary to do so succinctly. We also believe that our critic taxonomy is useful for guiding the critic developer towards realizing robust critic capabilities by comparing and contrasting different critic dimensions. We have applied our taxonomy to thirteen exemplar tools that have critic support along with our own critic specification meta-tool, the design of which has been influenced by the development of our taxonomy. The mapping of the tools to our critic taxonomy shows that the practice of critics is supported by the critic taxonomy.

In future work, we plan to explore in more detail the dependency between the various characterizing groups and their values. This will provide more design guidance to critic tool designers and assist critic tool end users in identifying and capturing their key critic requirements. As a longer-term goal we plan to incorporate this guidance into Marama Critic Designer in the form of meta-critics, to assist DSVL tool developers to choose appropriate critic implementation approaches for their tools. We will also apply the taxonomy to other computer-supported critics, possibly identifying further group elements and group and element dependencies.

7. Acknowledgment

The authors gratefully acknowledge the financial assistance of: the FRST Software Process and Product Improvement project; the Ministry of Higher Education Malaysia; and the Universiti Putra Malaysia in undertaking this research.

8. References

[1] "ArchStudio", http://www.isr.uci.edu/projects/archstudio/, 2013.
[2] "ArgoUML", http://argouml.tigris.org/, 2013.
[3] "MetaCase", http://www.metacase.com/, 2013.
[4] "Rational Rose Enterprise", http://www-03.ibm.com/software/products/us/en/enterprise/, 2013.
[5] "RSSE", https://sites.google.com/site/rsseresearch/tools, 2013.
[6] Ali, N.M., Hosking, J., and Grundy, J., A taxonomy of computer-supported critics, in Proceedings of the 2010 International Symposium in Information Technology (ITSIM2010), Kuala Lumpur, Malaysia, 15-17 June 2010, p. 1152-1157.
[7] Ali, N.M., et al., End-user oriented critic specification for domain-specific visual language tools, in Proceedings of the IEEE/ACM international conference on Automated Software Engineering. 2010, ACM: Antwerp, Belgium. p. 297-300.
[8] Ali, N.M., et al., Template-based critic authoring for domain-specific visual language tools, in Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing, Corvallis, Oregon, USA, 20-24 Sept 2009, p. 111-118.
[9] Bergenti, F. and Poggi, A., Improving UML designs using automatic design pattern detection, in Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE 2000), 2000, p. 336--343.
[10] Bezivin, J., and Jouault, F., Using ATL for checking models. Electronic Notes in Theoretical Computer Science, 2006, (152), p. 69-81.
[11] Blackwell, A. F., and Green, T. R. G. (1999). Investment of attention as an analytic approach to cognitive dimensions. In *Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG-11)* (pp. 24-35).
[12] Bolcer, G. A., User interface design assistance for large-scale software development. Automated Software Engineering, 1995, 2(3), p. 203-217.
[13] Bosansky, B., and Lhotska, L., Agent-Based Process-Critiquing Decision Support System, in Proceedings of the 2nd International Symposium on Applied Sciences in Biomedical and Communication Technologies, (ISABEL 2009), p. 1-6.
[14] Burke, R., Hybrid Recommender Systems: Survey and Experiments. User Modelling and User-Adapted Interaction, November 2002, vol. 12, no. 4, p. 331-370.
[15] Chen, L., and Pu P., Interaction design guidelines on critiquing-based recommender systems. User Model User-Adapted Interaction.2009. 19(3): p. 167-206.

[16] Chen, L., and Pu P., Critiquing-based recommenders: survey and emerging trends. User Model User-Adapted Interaction. 2012, 22(1-2): p.125-150.

[17] Chesnevar, C.I., Maguitman, A.G., and Simari, G.R., Argument-Based Critics and Recommenders: A Qualitative Perspective on User Support Systems. Data and Knowledge Engineering. Nov 2006, vol. 59, issue 2, p. 293-319.

[18] Cheung, W.K., 2012, priv. comm.

[19] Coelho, W., and Murphy, G., ClassCompass: A software design mentoring system. ACM Journal on Educational Resource in Computing, 2007. 7(1): p. 1-18.

[20] Cook, S., et al., Defining the context of OCL expressions. Lecture Notes in Computer Science. 1999. R. France And B. Rumpe (Eds). p. 372-383.

[21] Dashofy, E. M., Hoek, A. V. D., and Taylor, R. N. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2005, *14*(2), p. 199-245.

[22] de Souza, C. R. B., et al., A group critic system for object-oriented analysis and design, in Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering, 2000, p. 313 - 316.

[23] de Souza, C. R. B., et al., Using critiquing systems for inconsistency detection in software engineering models, in Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE 2003), p. 196-203.

[24] Fischer, G., A critic for LISP in the10th International Joint Conference on Artificial Intelligence, 1987, p.177-184.

[25] Fischer, G., Human-computer interaction software: lessons learned, challenges ahead. IEEE Software, 1989.6(1), p. 44-52.

[26] Fischer, G., and Girgensohn, A., End-user modifiability in design environments, in Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people (CHI'90), Jane Carrasco Chew and John Whiteside (Eds.). ACM, New York, USA, p. 183-192.

[27] Fischer, G., Lemke, A. C., and Mastaglio, T., Critics: an emerging approach to knowledge-based human computer interaction. International Journal of Man-Machine Studies, 1991.35:p. 695-721.

[28] Fischer, G., et al., The role of critiquing in cooperative problem solving. ACM Transactions on Information Systems, 1991. 9(3), p. 123-151.

[29] Fischer, G., and Mastaglio, T., A conceptual framework for knowledge-based critic systems. Decision Support Systems, 1990. 7: p. 355-378.

[30] Fischer, G, Context-Aware Systems- The 'Right' Information, at the 'Right' Time, in the 'Right' Place, in the 'Right' Way, to the 'Right' Person, in Proceedings of AVI'12, Capri Island, Italy, May 21- 25, 2012.

[31] Florijn, G., RevJava-Design critiques and architectural conformance checking for Java Software: Software Engineering Research Centre.2002. http://www.imamu.edu.sa/dcontent/IT_Topics/java/whitepaper_revjava.pdf

[32] Fu, M.C., Hayes, C.C., and East, E.W., SEDAR: Expert critiquing systems for flat and low-slope roof design and review, Journal of Computing in Civil Engineering, 1997. 11(1), p. 60-68.

[33] Gertner, A. S., and Webber, B. L., TraumaTIQ: Online decision support for trauma management. IEEE Intelligent Systems, 1998. p. 32-39.

[34] Green, T. R. G., and Blackwell, A. F., Cognitive dimensions of information artefacts: a tutorial, 1998. http://128.232.0.20/~afb21/CognitiveDimensions/CDtutorial.pdf

[35] Grundy, J., et al., Marama: an eclipse meta-toolset for generating multi-view environments in Proceedings of the International Conference on Software Engineering, 2008. p. 819-822.

[36] Irandoust, H., Critiquing systems for decision support. Technical Report. No. DRDC Valcartier TR 2003-321: Defence Research and Development Canada, 2006.

[37] Kelly, S., Lyytinen, K., and Rossi, M., MetaEdit+: A fully configurable multi-user and multi-tool CASE and CAME environment. Advanced Information Systems Engineering, 1996. Volume 1080/1996, p. 1-21.

[38] Kleppe, A., and Warmer, J., The Semantics of the OCL action clause. Object Modelling with the OCL, Lecture Notes in Computer Science, 2002, Vol. 2263, p. 213-227.

[39] Knauss, E., Luebke, D., and Meyer, S., Feedback-driven requirements engineering: the heuristic requirements assistant, in Proceedings of the IEEE 31st International Conference on Software Engineering, 2009. p. 587 - 590.

[40] Knauss, E., 2010, priv.comm.

[41] Lee, C., et al., Web-based essay critiquing system and EFL students ' writing: a quantitative and qualitative investigation, Computer Assisted Language Learning, 2009, Vol. 22, No. 1, p. 57-72.

[42] Lemke, A. C., and Fischer, G., A cooperative problem solving system for user interface design, in Proceedings of the Eight National Conference on Artificial Intelligence, 1990, p. 479-484.

[43] McCarthy, K., et al., Experiments in Dynamic Critiquing in Proceedings of the International Conference on Intelligent User Interfaces (IUI ’05), 2005, p. 175-182.

[44] McCarthy, K., et al., Group Recommender Systems: A Critiquing Based Approach, in Proceedings of the International Conference on Intelligent User Interfaces, 2006, p. 267-269.

[45] Miller, P., Expert Critiquing Systems: Practice-based Medical Consultation by Computer. New York: Springer-Verlag, 1986.

[46] Nakakoji, K., Case-Deliverer: Retrieving Cases Relevant to the Task at Hand. In Selected papers from the First European Workshop on Topics in Case-Based Reasoning (EWCBR-93), Stefan Wess, Klaus-Dieter Althoff, and Michael M. Richter (Eds.), Springer-Verlag, London, UK, p. 446-457.

[47] Nakakoji, K., Reeves, B.N., Aoki, A., Suzuki, H., and Mizushima, K., eMMaC: Knowledge-based colour critiquing support for novice multimedia authors, in Proceedings of the third ACM international conference on Multimedia (MULTIMEDIA ' 95), 1995, p. 467-476

[48] Oh, Y., Do, E. Y.-L., and Gross, M. D., Intelligent critiquing of design sketches. Making Pen-based Interaction Intelligent and Natural, (Eds, Randall Davis, JL, Stahovich, T, Miller, R and Saund, E), AAAI Press, Arlington, Virginia, 2004, pp 127-133.

[49] Oh, Y., Gross, M. D., and Do, E. Y.-L., Computer-aided critiquing system, in Proceedings of the Computer Aided Architectural Design and Research in Asia (AADRIA).2008, p. 161-167.

[50] Oh, Y., et al., Constraint-based design critic for flat-pack furniture design, in Proceedings ofthe17th International Conference on Computers in EDUCATION. 2009, p. 19-26.
[51] Pisan, Y., et al., Submit! A Web-Based System for Automatic Program Critiquing in Proceedings of the Fifth Australasian Computing Education Conference (ACE2003), 2003, p. 59-68.
[52] Pisan, Y, 2012, priv. comm.
[53] Qiu, L., and Riesbeck, C.K., An incremental model for developing educational critiquing systems: experiences with the Java Critiquer. Journal of Interactive Learning Research, 2008, 19(1): p. 119-145.
[54] Qiu, L., and Riesbeck, C. K., Facilitating critiquing in education: the design and implementation of the Java Critiquer, in Proceedings of the International Conference on Computers in Education (ICCE) 2003.
[55] Reilly, J., et al., Incremental Critiquing. Knowledge-Based Systems, 2005. 18:p. 146-151.
[56] Riesbeck, C.K., and Dobson, W., Authorable critiquing for intelligent educational systems, in Proceedings of the 3rd International Conference on Intelligent User Interfaces. 1998, p. 145-152.
[57] Robbins, J. E., Design critiquing systems. Technical Report. Irvine: Department of Information and Computer Science, University of California. 1998.
[58] Robbins, J. E., and Redmiles, D. F., Software architecture critics in the Argo design environment. Knowledge-Based Systems, 1998. 11(1), p. 47-60.
[59] Robbins, J. E., and Redmiles, D. F., Cognitive support, UML adherence, and XMI interchange in ArgoUML. Journal of Information and Software Technology, 2000. 42(2), p. 79-89.
[60] Silverman, B.G., Expert critics: operationalizing the judgement/decision making literature as a theory of "bugs" and repair strategies. Knowledge Acquisition, 1991, 3(2), p. 175-214.
[61] Silverman, B. G., Survey of expert critiquing systems: practical and theoretical frontiers. Communications of the ACM, 1992, p. 35(4).
[62] Silverman, B. G., and Mehzer, T. M., Expert Critics in Engineering Design: Lessons Learned and Research Needs. AI Magazine, 1992. 13:p. 45-62.
[63] Sumner, T., Bonnardel, N., and Kallak, B.H., The cognitive ergonomics of knowledge-based design support systems, in Proceedings of the SIGCHI conference on Human factors in computing systems (CHI'97), ACM, New York, USA, p. 83-90.
[64] Stolze, M., Visual critiquing in domain oriented design environments: showing the right thing at the right place, in Proceedings of the Artificial Intelligence And Design. 1994, p. 1-16.
[65] Tianfield, H., and Wang, R., Critic system- towards human-computer collaborative problem solving. Artificial Intelligence Review, 2004. 22: p. 271-295.
[66] Trochim, W. M. K., Outcome pattern matching and program theory. Journal of Evaluation and Program Planning, 1989. 12(4): p. 355-366.
[67] Tyugu, E., Algorithms and architectures of artificial intelligence. Frontiers in AI and Applications, 2007. Vol. 159, IOS Press.
[68] Zhu, N., et al., Pounamu: A meta-tool for exploratory domain-specific visual language tool development, Journal of Systems and Software, 2007. 80(8), p. 1390-1407.

Norhayti Md Ali Norhayati Mohd Ali received the PhD degree in computer science from the University of Auckland, New Zealand in 2011. She is a senior lecturer at the Information System Department, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia. Her research interests include critiquing tools in software engineering, recommendation systems in software engineering, software engineering education, and human-computer interaction.

John Hosking is Dean and Director of the College of Engineering and Computer Science at the Australian National University in Canberra Australia. John obtained his BSc and PhD in Physics from the University off Auckland then joined the Department of Computer Science at the University of Auckland working his way through to full Professor before taking up his current position at the ANU. John's research interests are in software engineering, visual languages, and knowledge visualisation. He is a MemIEEE and a Fellow of the Royal Society of New Zealand.

John Grundy is is Professor of Software Engineering and Head of Computer Science and Software Engineering at the Swinburne University of Technology. He has published extensively in areas including Automated Software Engineering, Cloud Computing, Model-driven Development, Software Methods and Tools, Software Architectures and Visual Languages.