Usability and Suitability Survey of Features in Visual IDEs for Non-Programmers

Jean Michel Rouly
George Mason University
jrouly@masonlive.gmu.edu

Jonathan D. Orbeck
University of Alabama
jdorbeck@crimson.ua.edu

Eugene Syriani
University of Montreal
syriani@iro.umontreal.ca

Abstract

Software tools and working environments differ drastically from one domain to another. The software engineering domain is entertained by a diversity of rich integrated development environments (IDEs) that aim at simplifying the tasks and reducing the efforts of a programmer. Other domains are unfortunately not as cherished. In this paper, we survey twenty-five visual IDEs used in non-programming domains and evaluate how usable and fit they are for their domain. The goal of this research is to determine what features are needed in different domains and how they should be presented to the domain user.

Keywords Integrated Development Environments; Visual Languages; Domain-Specific Languages; Usability Study

1. Introduction

Software being a ubiquitous technology, it is nowadays being used in a wide spectrum of domains: music, teaching, arts, engineering. The proliferation of visual Integrated Development Environments (IDEs) has helped better assist these domain users who are unfamiliar with programming. Common requirements to IDEs in general often include [8]: uniformity and consistency across the different tools it provides, an interactive user interface for performing the different tasks, ability to inspect a well-defined state of the system being developed (*e.g.,* debugging), integrate control of versions of the system in an individual or collaborative environment, and ensure a manageable development process.

Previous studies have surveyed IDEs, but they were mostly focused on the programming domain [5, 8, 10]. In this study, we focus on IDEs used in non-programming domains where the underlying language is visual (*e.g.,* diagrams), as opposed to textual (*e.g.,* source code). Often these IDEs provide the user with information about the syntax of the supported language(s) or otherwise provide features that integrate with any constraints of the visual language. Components and behaviors of the IDE interface can greatly affect the overall usability of an IDE as well as its suitability to its supported language. Consequently, it is important to utilize as many techniques as possible to ensure both overall interface usability and suitability to the target language. Several works have looked at metrics for measuring the usability of software [2, 7]. We have adapted these metrics to specifically look at visual IDEs in different domains.

The goal of this study is to evaluate existing IDEs in a variety of domains to make the developers of cross-domain generic IDEs aware of the different features needed to make their tools more usable and suitable by their target users. generative product line approaches, such as domain-specific modeling (DSM) [11], would highly benefit from such a study.

Section 2 describes the methodology and features covered by our survey. Section 3 summarizes the results of the study for each IDEs surveyed. In Section 4, we discuss interpretation and limitations of the survey, and conclude in Section 5.

2. Methods

In the following, we define the terminology used in the study, the process and criteria for selecting a set of IDEs for the study, the definition of a set of novel visual IDE interface features, and the process of evaluation taken for each IDE.

2.1 Terminology

Fig. 1 is a feature model describing the features we considered for the visual IDEs. They are defined in Section 2.3.

The *workspace* of an IDE is sometimes referred to as a *canvas* and is an area within the IDE interface with which the operator directly interacts to create and store content. Generally, the workspace does not contain a listing of available tools. The *supported language* of an IDE is the visual

PLATEAU '14, October 21, 2014, Portland, OR, USA..

http://dx.doi.org/10.1145/2688204.2688207

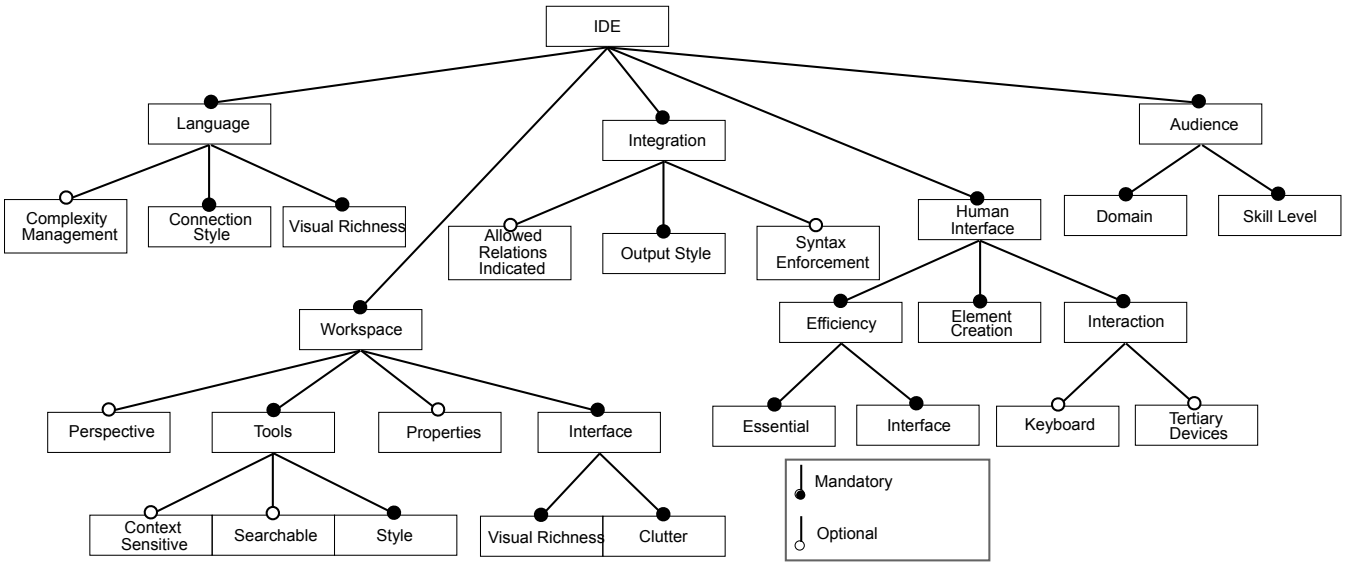

Figure 1. Visual IDE feature diagram model

language the IDE allows its user to work in. The IDE includes features and tools to support development within this language. *Elements* within the language are syntactic units or lexemes represented visually within the IDE workspace. Similarly, *relations* or *connections* are syntactic relationships between elements represented visually within the workspace [4].

2.2 IDE Selection

We decided to select IDEs based on their support for a visual language as the primary language of development. A subsidiary criteria for selection included popularity of the IDE within its domain as well as representation from a variety of development domains. Thus this survey considers IDEs from eight different domains: 3D modeling, animation, modeling, music, prototyping, simulation, visual software development (*e.g.,* education, mobile development, etc.), and business workflow.

2.3 Feature Definition

The following enumerates a unique set of features based on existing literature and emergence of similarities and differences between the IDEs considered. Features are categorized under the following five headings.

2.3.1 Audience Features

The audience of an IDE refers generally to the target populations for which the IDE is intended.

Domain The domain of an IDE refers to the domain or field of knowledge under which the interface falls. This is a nominal variable where example values might include *General*, *Modeling*, *Software*, etc. The latter *General* indicates that the IDE is of general purpose and can be applied within any number of different fields to the same extent.

Skill Level The skill level of an IDE describes what level of domain expertise is expected of users for using it. This is a nominal variable where possible values are *Novice*, *Intermediate*, *Expert*, and *General*. The latter indicates that the IDE offers a powerful set of advanced features while maintaining components that emphasize accessibility and ease of use.

2.3.2 Chrome Features

The chrome of an IDE is the total set of all graphical user interface (GUI) components external to the workspace. This includes every tool, menu, button, or other user interface component not contained within the workspace area.

General Operations Many IDEs provide support for a specific subset of common operations. Murphy et al. [13] define a list of the top 10 IDE features most frequently utilized by developers. We reproduced it here as a set of Boolean subvariables that indicates whether the IDE in question supports the use of that particular operation. Due to space limitations, we report only the sum of the *true* values. The operations are: *delete* a syntactic element in the workspace, *save* to export a model to storage media, *cut*/*copy*/*paste* an existing syntactic element in the workspace, *undo* the user's most recent action. *Content assist* provides suggestions or completion for elements and *refresh* loads contents of workspace and interface dashboard elements from storage media and update display if necessary. *Show view* opens and displays a new tool in the interface, and *next word* moves active selection to the next element according to some natural ordering, for example as a result of searching.

Context Sensitive Tools Any interface component which changes visibly or is generated anew depending on the context of selected elements within the workspace is context

sensitive, *e.g.,* a popup context menu that appears when an element is clicked which provides tools or information about the clicked element. This Boolean variable indicates whether context sensitive tools are supported.

Degree of Interface Visual Richness This describes the extent to which an IDE utilizes visual variables to increase the visual discrimination of tools it offers as advised by Tufte's guidance [14] and Moody's specifications [12]. This compound variable is composed of eight sub-variables, each being a Boolean measure determining whether or not the described visual variable is utilized in the interface to distinguish between available tools in the IDE. Due to space limitations, we report only the sum of the *true* values. The variables include: *icons* [4, 12] as images contained in a border of a standard size and shape, *shapes* [12], tool *size* [12], *color* [12], *text* or typographic variation [12], *texture* [12] as shading or shadows, *brightness* of a color (*i.e.,* its perceived luminosity) [12], and *organizational coherence* [2] to determine if components with related purpose are visually grouped together in the interface.

Multiplicity of Perspectives A perspective is defined as a visual configuration of tools in the GUI for the purpose of accomplishing a distinct task as part of a distinct process, *e.g.,* debugging, file browsing, manipulation of element details. Some IDEs, *e.g.,* MST Workshop, even support multiple languages or domains through the use of perspectives by offering entirely different feature sets. This metric measures the number of available predefined interface perspectives available to the user, with values greater than zero.

Object Properties Window This is an interface component that displays the properties of an element in the workspace, typically to view and modify properties of model elements. For example, the IDE GNU Radio Companion (GRC) allows elements to take user-defined values for various properties (*e.g.,* frequency, amplitude). These properties can be manipulated when the user explicitly requests the properties dialog window for that element. This is a nominal variable whose values are: *None* if no object properties window is available, *Omnipresent* if such window is always present, allowing contents to update contextually, and *Manual* if such window requires user interaction to bring forward.

Searchable Toolspace This is a Boolean variable that indicates whether the total set of available tools, components, or actions offered by the IDE can be searched through by name or keyword, *e.g.,* Caméléon which provides the user a search box to navigate its library of predefined available elements.

Toolbar Styles These refer to the set of GUI component idioms employed by the IDE [6]. This nominal variable can take combinations of multiple values, such as: *Icons*, *Menus*, *Ribbons*, *Trees*.

Visual Clutter The clutter of an interface is the number and organization of tools available on the screen versus the amount of workspace provided by the IDE [3, 6]. If the IDE offers no method for tool organization or if there is an immense amount of tools visible at once, then the IDE is likely to be visually cluttered. IDEs which, for example, simply list available tools in rows of toolbars (*e.g.,* Fig. 2) do not offer options for interface organization and represent high visual clutter. Visual clutter is a nominal variable and takes the values *Low*, *Medium*, or *High*. Section 2.4 includes details on the proper evaluation of this qualitative feature.

2.3.3 Human Interface Features

The human interface features of an IDE include aspects of the software interface that affect how the user interacts with the IDE, either mechanically (*e.g.,* through physical devices and media) or mentally (*e.g.,* the mental load required of the user to operate the IDE).

Essential Efficiency The essential efficiency of an IDE measures the level to which the system automates tasks for the user. It is calculated as the ratio of the number of steps in a concrete use case C to the number of steps in the essential use case E as depicted in Equation 1. A concrete use case describes the steps in the specific IDE to perform the same tasks as in an essential use case. This metric, unlike those for automation put forth in [15], does not require user experience reports and can be measured through simple use case analysis. Section 2.4 gives more details on the essential use cases considered.

$$1 - \frac{C}{E} \tag{1}$$

Interface Efficiency The interface efficiency of an IDE is a concept related to the productivity of an interface. It measures the number of physical actions (including keystrokes, mouse clicks, and fine mouse movements) A required of the user to complete a task compared against the number of abstract steps in the essential use case, as depicted in Equation 2.

$$1 - \frac{A}{E} \tag{2}$$

This metric is different from the Essential Efficiency proposed in [2] because it studies physical user actions instead of concrete task steps. Also note that this variable can take negative values, indicating that the number of physical actions required to complete an essential use case exceeds the number of abstract steps in the use case.

Keyboard Use This refers to the extent to which an IDE utilizes the use of a keyboard. This can range from a complete absence of any keyboard actions to providing certain actions which only a keyboard can perform. Keybindings (optional or required) are a common way to provide keyboard interactivity.

This nominal variable can take the values: *None* when no keyboard use is supported, *Simple* when the keyboard is

used only for typing annotations, properties, or comments, *Optional* when the option to use the keyboard to execute some actions is present, but these actions can also be completed using a mouse, and *Required* when there are actions that can only be completed through the use of the keyboard, no mouse equivalent is available.

Mode of Element Creation This describes the process through which the user creates elements in the IDE. The *Drag n Drop* process refers to a single mouse press event followed by a dragging motion of the mouse and completed when the mouse button is released, *e.g.,* selecting and dragging a template into the canvas to create a new element. The *Point n Click* process utilizes a single mouse click to indicate a selection followed with subsequent mouse clicks elsewhere to define placement, *e.g.,* clicking a tool symbol to select active element type and then clicking in the workspace to create elements at specific positions. This nominal variable can take one of four possible values created by combining *Drag n Drop* or *Point n Click* with the multiplicity of the action: either *(1:1)* or *(1:n)*. The former multiplicity indicates that a single element is created for each action, while the latter indicates that multiple can be created after the action.

Tertiary Interface Devices This nominal variable describes any third party human interface devices which can be used to interact with the IDE. This could include audio devices such as MIDI keyboards or microphones, mobile integration, etc. Variable values are the type of tertiary devices allowed for the IDE. AudioMulch and Max, for example, allow for the integration of microphones as data input devices. AppInventor supports exporting to mobile devices.

2.3.4 Integration Features

Integration is the manner with which the IDE integrates with the visual language it supports. This includes any visual representation of language syntax or semantics, as well as any tools to assist the user with understanding the supported language.

Allowed Relations Indicated This refers to an IDE's ability to emphasize possible syntactically correct connection points. This is often demonstrated with either the highlighting of allowed relations or the dimming of impossible relations, *e.g.,* Caméléon which color codes available connection endpoints when the operator begins creating a connection. Boolean values indicate whether the IDE supports this feature or not.

Output Generation Style This nominal variable describes the mode with which the IDE renders and displays output to the user. It is a dual axis nominal variable, measuring whether output is direct or indirect as well as live or caused by a trigger. *Direct/indirect* describes whether the user directly modifies output or acts via a layer of abstraction (*e.g.,* via a model as in Grasshopper, or directly as in Blender). *Live/trigger* describes whether output is generated and displayed live or after some event triggered by the user, *e.g.,* a compilation or build request.

Syntax Enforcement The level of syntax enforcement of an IDE describes the mode with which the IDE enforces its supported language's syntax requirements, if at all. The nominal value *Explicit* indicates that the IDE explicitly enforces syntax requirements by indicating to the user the presence of any syntax errors, *e.g.,* error popups informing the user that their creation has an error. *Implicit* enforcement indicates that the IDE does not allow syntactically illegal operations to occur in the first place by means of some structural mechanism, *e.g.,* snapping puzzle piece elements which do not snap to illegal connections. The value *None* indicates that the IDE does not support syntax enforcement and the user is required to review their models' syntax manually.

2.3.5 Language Syntax Features

Language syntax variables describe properties of the supported visual language syntax. While not strictly components of the IDE, they are intimately tied to the overall style of the IDE and thus included in this study.

Complexity Management Any characteristics or features of the visual language that serve to reduce the complexity of that language. Reducing complexity refers specifically to decreasing the level of *diagrammatic complexity* while maintaining information transfer to the user [12]. This can be implemented variously, *e.g.,* modularization of large projects into files or hierarchical abstraction into levels of detail.

This nominal variable can take one of three possible values. *Modularization* indicates that large systems within the language are divided into smaller subsystems to reduce complexity [12]. *Hierarchy* indicates that systems within the language can be represented at different levels of detail [12]. *None* indicates that the visual language does not support complexity management functionality.

Connection Style A language's connection style refers to the manner with which connections between elements are displayed. This dual axis nominal variable measures connections as *overlapping vs. linked* as well as connection sources as *point vs. region* based [4]. The former axis describes the visual representation of the connections, while the latter describes how links are connected. MST Workshop, for example, allows the user to overlap points on elements in order to indicate connection, whereas Grasshopper requires specific linking between points on elements. If the supported language does not fall on either of these axes, it is not connection based [4] and this feature takes the value *Geometric*.

Degree of Language Visual Richness This describes the extent to which a language utilizes visual variables to increase the visual discrimination of its elements. This is a compound variable composed of ten sub-variables, each being a Boolean measure similarly to the IDE's homologue. The same sub-variables are icons, shapes, size, color, text,

texture, and brightness. Additionally, *orientation*, *horizontal* and *vertical positioning* [12] are added to further distinguish between elements.

2.4 IDE Evaluation

With IDE feature definitions established, we evaluate each IDE by measuring the different metrics according to each feature. Most features only require simple classification or binary evaluation tasks. However, three features require more intensive evaluation.

2.4.1 Evaluation of Visual Clutter

Evaluating the visual clutter of an IDE requires qualitative user feedback. We used Amazon.com's Mechanical Turk (MTurk) as a crowd-sourcing platform to perform the user study and gauge opinions of visual clutter. We generated three unique screenshots of each IDE and darkened the workspace area to remove attention from diagrammatic complexity. See Fig. 2 as an example image provided on MTurk. These images were distributed as MTurk Human Intelligence Tasks (HITs), requiring five unique evaluators per HIT. On average, each evaluator spent about one minute studying and rating an image on a scale of 1 (low clutter) to 5 (high clutter). Each HIT rewarded its evaluator with $0.02. IDE rating values were retrieved and averaged between each reviewer for one screenshot and then each screenshot for one IDE. Upon completion, the HITs were rated by 12 anonymous workers in total.

Because multiple reviewers rated each image, we performed an Inter-Rater Reliability (IRR) measure to ensure agreement across reviewers. Using the R statistical library *irr* we perform a two-way agreement average-measure Intra-Class Correlation (ICC). The result, ICC = 0.648, is within the "good" range of significance [1, 9]. This ICC value indicates that the reviewers were, in general, in agreement about their ranking of interface clutter. Note that there were more than five reviewers participating total, despite five sets of reviews (*i.e.,* the experimental design is not "fully crossed" [9]). We argue that this is not significant, however, because the five sets of reviewers are disjoint sets, acting as entirely independent actors.

2.4.2 Evaluation of Efficiency

The evaluation of both interface and essential efficiency involved the creation of use cases for each IDE and subsequent evaluation of the corresponding concrete use cases. We developed two essential use cases common to all IDEs and a third one that differed slightly for each. The two simple essential use cases were *Open File* and *Create Element*, three and four steps respectively. The third, complex essential use case varied between IDEs depending on applicability: *Create and Link Elements* (13 steps), *Create Element and Transform* (10 steps), *Print an Integer in Piet* (12 steps), *Print a Character in TouchDevelop* (9 steps). The three essential use cases were ordered of increasing task complexity.

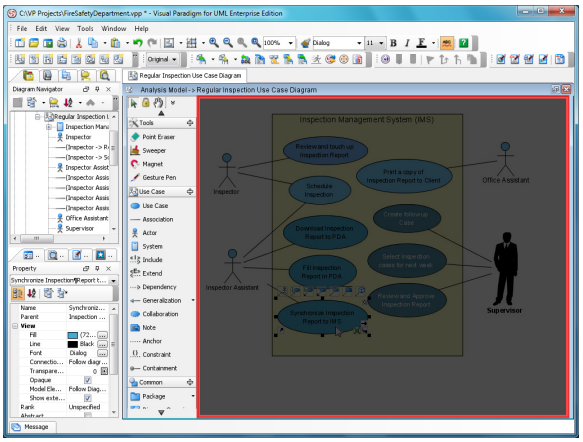

Figure 2. Sample IDE screenshot provided on Mechanical Turk (Visual Paradigm)

Each essential use case was then manually performed within the IDEs to generate corresponding concrete use cases. Every physical action required for completion was tracked: keystrokes, mouse clicks, and fine mouse movements.

After all concrete use cases were completed, we noticed that the results from the first two (less complex) use cases did not vary significantly enough across IDEs. We therefore report only the results from the more complex use case for each IDE.

3. Results

Tables 1–3 outline the results of the evaluation of the 25 IDEs we considered in this study. We give further explanations below for each in alphabetical order. What follows is based on the measured evaluation results accompanied with our own opinion of each tool.

Alice3 One of the earliest educational interfaces present in this study, Alice has influenced a great deal of later IDEs, including Stencyl, AppInventor, and Scratch to name a few. Designed primarily as an educational programming environment, Alice only targets users with a low level of skill in its domain. It makes up for this loss in accessibility, however, with its wide, almost universal, support of common IDE features as well as context-sensitive tooling. Alice3, the current release, provides a medium level of visual richness in its interface chrome, but boasts one of the highest essential efficiency values. This efficiency value is achieved through the style of visual syntax proper to Alice and its successors. However, despite greatly reducing operator mental load, its visually cluttered design only manages a near-neutral interface efficiency. The introduction of optional keybindings is a redeeming factor for Alice3 and, like most educational programming IDEs, Alice3 includes simple modular complexity management and a visually appealing level of language visual richness.

IDE	Features	Context	ToolRichness	Perspectives	Properties	Searchable	ToolStyle	Clutter
Alice3	8	Yes	6	2	Omnipresent	No	Tabs	3.40
AToMPM	4	No	5	1	Manual	No	Modular	2.60
AudioMulch	7	Yes	6	2	Manual	No	Tree, Windows	3.13
Blender	7	Yes	7	Multiple	Omnipresent	No	Multiple	4.00
Caméléon	3	Yes	5	1	Manual	Yes	Tree	2.07
EMF	9	Yes	5	Multiple	Manual	Yes	Tree, Icons	4.07
GNU Radio Companion	6	No	4	1	Manual	Yes	Icons	2.80
Grasshopper 3D	7	Yes	4	1	None	Yes	Ribbons	2.80
Max	7	No	4	2	Manual	Yes	Tree, Windows	2.73
MetaEdit+	7	Yes	4	3	Manual	No	Icons, Windows	1.93
MIT AppInventor2	3	Yes	4	2	Omnipresent	No	Drawers	3.27
MST	6	No	3	1	Manual	No	Icons	2.40
Piet Creator	4	No	7	1	None	No	Icons	2.13
Scratch	3	No	6	2	None	No	Drawers	3.53
SimuLink	7	No	3	1	Manual	Yes	Icons, Tree	3.80
Stencyl	6	Yes	8	Multiple	Manual	Yes	Tabs, Icons	2.73
Tersus	8	No	3	1	Omnipresent	No	Drawers	3.20
TouchDevelop	6	Yes	6	1	None	No	Icons	3.60
UMLet	6	No	3	1	Omnipresent	No	Palette	3.00
VioletUML	6	No	4	1	Manual	No	Drawers	2.47
VisSim	8	No	4	1	Manual	Yes	Tree, Menu	3.33
Visual Paradigm	7	Yes	5	1	Omnipresent	No	Icons, Menu	3.67
Visual Use Case	3	No	4	7	Manual	No	Icons, Tree	2.67
WebRatio	8	No	4	3	Omnipresent	No	Icons, Menu	3.93
YAWL	5	No	5	2	Omnipresent	No	Icons	2.27

Table 1. Measure of the chrome of IDEs.

IDE	Audience		Human Interface				
	Domain	Skill	EEfficiency	IEfficiency	Keyboard	Mode	Devices
Alice3	Animation	Novice	0.46	0.08	Optional	Drag n Drop (1:1)	None
AToMPM	Modeling	Intermediate	0.23	0.15	Optional	Point n Click (1:n)	None
AudioMulch	Music	Expert	0.31	0.31	Simple	Drag n Drop (1:1)	Keyboards
Blender	3D Modeling	Expert	0	-0.2	Required	Point n Click (1:1)	None
Caméléon	Prototyping	Intermediate	0.23	0.31	Required	Drag n Drop (1:1)	None
EMF	Software	Expert	0.23	0.62	Optional	Point n Click (1:1)	None
GNU Radio Companion	Software	Intermediate	0.38	0.62	Optional	Drag n Drop (1:1)	None
Grasshopper 3D	3D Modeling	Intermediate	0.31	0.31	Optional	Point n Click (1:1)	None
Max	Music	Expert	0.15	0.15	Simple	Drag n Drop (1:1)	Microphones
MetaEdit+	Modeling	Intermediate	0.31	-0.08	Simple	Point n Click (1:1)	None
MIT AppInventor2	Software	Novice	0.46	0.54	Simple	Drag n Drop (1:1)	Mobile
MST	Simulation	Intermediate	0.38	0.38	Simple	Point n Click (1:1)	None
Piet Creator	Software	General	0.17	-0.33	None	Point n Click (1:n)	None
Scratch	Software	Novice	0.46	0.54	Simple	Drag n Drop (1:1)	None
SimuLink	Simulation	Expert	0.31	0.15	Simple	Drag n Drop (1:1)	None
Stencyl	Software	Novice	0.46	0.54	Simple	Drag n Drop (1:1)	Mobile
Tersus	Software	Intermediate	0.31	0.15	Simple	Point n Click (1:1)	None
TouchDevelop	Software	Novice	0.44	0.33	None	Point n Click (1:1)	Mobile
UMLet	Modeling	General	0.31	-0.15	Simple	Drag n Drop (1:1)	None
VioletUML	Modeling	Intermediate	0.15	0.15	Simple	Point n Click (1:n)	None
VisSim	Simulation	Expert	0.31	0.31	Simple	Drag n Drop (1:1)	None
Visual Paradigm	Modeling	Intermediate	0.23	0.23	Simple	Drag n Drop (1:1)	None
Visual Use Case	Modeling	Intermediate	0.54	0.38	Simple	Point n Click (1:1)	None
WebRatio	Software	Intermediate	0.23	0.23	Simple	Point n Click (1:1)	None
YAWL	Workflow	Expert	0.31	0.31	Simple	Point n Click (1:n)	None

Table 2. Measure of the human interface of IDEs and their intended audience.

IDE	Integration			Language Syntax		
	RELATIONS	OUTPUT	SYNTAX	COMPLEXITY	CONNECTION	LANGUAGE RICHNESS
Alice3	No	Direct Triggered	Implicit	Modularization	Overlapping Regions	8
AToMPM	No	Indirect Triggered	Explicit	None	Linked Regions	6
AudioMulch	Yes	Indirect Live	Implicit	None	Linked Points	3
Blender	No	Direct Live	Implicit	Modularization	Geometric	9
Caméléon	Yes	Direct Live	Explicit	Hierarchy	Linked Points	5
EMF	No	Indirect Triggered	Explicit	Modularization	Linked Points, Linked Regions	5
GNU Radio Companion	No	Indirect Triggered	Explicit	None	Linked Points	2
Grasshopper 3D	No	Indirect Live	Explicit	None	Linked Points	6
Max	No	Indirect Live	None	Modularization	Linked Points	5
MetaEdit+	No	Direct Live	Implicit	Modularization	Linked Regions	5
MIT AppInventor2	No	Direct Triggered	Implicit	Modularization	Overlapping Regions	9
MST	No	Direct Live	None	None	Overlapping Points	4
Piet Creator	No	Indirect Triggered	None	None	Geometric	4
Scratch	No	Direct Triggered	Implicit	Modularization	Overlapping Regions	10
SimuLink	No	Indirect Live	None	None	Linked Regions	4
Stencyl	No	Direct Triggered	Implicit	Modularization	Overlapping Regions	10
Tersus	No	Direct Triggered	None	Hierarchy	Linked Points, Linked Regions, Geometric	3
TouchDevelop	No	Indirect Triggered	Explicit	Modularization	Geometric	4
UMLet	No	Direct Live	None	None	Overlapping Points	4
VioletUML	No	Direct Live	Explicit	None	Linked Regions	4
VisSim	No	Direct Live	None	Hierarchy	Linked Points	4
Visual Paradigm	No	Direct Live	Implicit	None	Linked Regions	2
Visual Use Case	No	Direct Live	None	Modularization	Linked Regions	3
WebRatio	No	Indirect Triggered	None	Hierarchy	Linked Regions, Geometric	4
YAWL	No	Direct Triggered	None	None	Linked Regions	4

Table 3. Measure of the integration and language syntax of IDEs.

AToMPM AToMPM is unique among the IDEs in this study in that it is built as a modeling framework specifically to develop domain-specific modeling IDEs. While AToMPM does not support a large number of popular IDE features or visual richness variables, its modular toolbar system helps to reduce visual clutter by loading only the features the user deems necessary. AToMPM allows for a small number of optional keybindings which can help increase utility, but essential efficiency and interface efficiency are only slightly above a one-to-one ratio with the essential use cases. Explicit syntax enforcement is provided which protects the user from potential illegal operations, but no complexity management system is in place to handle user mental load. Because AToMPM is used to develop IDEs it can be argued that resulting generated AToMPM tools may score differently, but overall AToMPM achieves roughly average performance compared to other IDEs in the study.

AudioMulch Although AudioMulch offers a wide array of tools and an in-depth interface ideal for professionals in the music industry, the overall complexity of the design greatly reduces the accessibility for anyone else. Nevertheless, it supports a large amount of popular IDE features and offers equally high essential and interface efficiency ratings. This is visually assisted by a relation-highlighting feature which also provides AudioMulch with an implicit syntax enforcement, both of which greatly aid the user in model creation. Unfortunately, there is no effort made to manage the high amount of complexity within the IDE and virtually all of the canvas elements look exactly the same, ultimately awarding AudioMulch with a low language visual richness score.

Blender Designed for a skilled target audience of experts in the domain, it is no surprise that Blender supports most common IDE features as well as context-sensitive tooling. Its high level of chrome visual richness and the large number of perspectives available relative to the average found in this study also contribute to a high quality interface. However, the fact that it offers so many features leads directly to the second highest observed value for visual clutter. Blender possesses no particular efficiency techniques, remaining around a perfect one-to-one relationship with the measured essential use cases. Its heavy use of the keyboard reduces accessibility to a wider audience, although the target skill level is already a limiting factor. Finally, Blender provides modularization complexity management through saving and duplication tools, along with one of the most visually rich languages observed.

Caméléon Intended for rapid, flexible visual prototyping of functional algorithms, Caméléon is relatively feature impoverished — it only supports three of the most popular IDE features. It does however boast a large number

of available tools that are conveniently searchable. The interface chrome of Caméléon employs five visual richness variables, slightly above the average count. Its low visual clutter interface provides a positive level of essential and interface efficiency. Interaction requires use of the keyboard for non-essential actions, specifically advanced navigation and zooming. Caméléon supports the user by interactively highlighting allowed syntactic relations and explicitly enforcing syntax requirements. More support is provided by its hierarchical complexity management system. Overall, Caméléon is an efficient, simple tool with a large library of functionality and a focus on supporting syntax requirements.

Eclipse Modeling Framework EMF is one of the most sophisticated IDEs in this study and demands an expert level of skill from target users. It is also the only IDE studied which supports all of the measured popular features, as any expert in the domain would likely come to expect. The Eclipse Modeling Framework (EMF) interface chrome supports five visual richness variables as well as a large number of predefined interface perspectives. The high level of supported perspectives, while perhaps intimidating, increases the overall power and utility of the interface. Additionally, the EMF tool space is searchable, which counterbalances the highest observed level of visual clutter. Despite being extremely cluttered, EMF has average levels of essential efficiency and the highest measured level of interface efficiency. The supported visual languages do not make use of more than five visual variables but do allow for complexity management. Overall, EMF is well suited for an expert user with high efficiency and several convenience features, but is too cluttered and not visually rich enough to support a wider audience of varied skill levels.

GNU Radio Companion GRC is a simple platform designed to aid the development of signal processing software without the need to understand or write code. It does, however, require a moderate level of skill in the domain. GRC supports slightly more than the average number of popular IDE features as well as the ability to search through available tools. The interface chrome of GRC only utilizes four visual variables while the supported language only employs two. Despite this, the interface is highly efficient, with the highest observed interface efficiency and a correspondingly high level of essential efficiency. It also supports optional keyboard use and explicit syntax checking. GRC is designed as an easy-to-use interface for non-programmers and manages to maintain simplicity while still offering a large amount of technical power.

Grasshopper 3D Though Grasshopper is able to provide the user with a relatively simple and easy-to-use interface, beginners would likely shy away from the complexity of its core functionality. Even so, it offers a high amount of popular IDE features and context sensitive tools, as well as the ability to search through its vast library of tools quickly and easily by name. Grasshopper also possesses very good efficiency techniques, maintaining both values at a more-than-decent level. In addition, the optional use of a keyboard is supported, offering functionality on another level to increase accessibility. However, complexity management is not supported at all and visual richness in both the language and tools are mediocre at best.

Max Its overly simple appearance can be misleading—it is very much designed for skilled users in the music development domain. Max supports several popular features and an average number of visual richness variables in both its chrome and supported language. Its use of a searchable toolspace is unique among the studied IDEs: in order to create almost any advanced element in the workspace, the user must search by name or description for the element. Only a small subset of its functionality can be reached otherwise. This simultaneously radically reduces visual clutter by hiding most elements from the user, while also drastically increasing mental overhead for users. Max offers a below average level of interface and essential efficiency, as well as a convenient modular complexity management system. Once mastered, Max can be a powerful and efficient tool for developing music and audio processing tools, but the amount to which the library of functionality is removed from the user can be intimidating and, ultimately, greatly decreases accessibility.

MetaEdit+ As for AToMPM, MetaEdit+ is a tool for creating domain-specific modeling environments. Aimed towards an intermediate level of users, the many different tasks and steps that go into the design of a simple model in MetaEdit+ can easily be overwhelming for newer users. These tasks are fortunately divided through modularization, allowing them to be much more manageable within the IDE. Furthermore, many of the popular IDE features as well as context sensitive tools are present in the interface, increasing the accessability even more. MetaEdit+ also holds the lowest clutter value and integrates an inplicit syntax enforcement, providing the user with a clear and easy-to-use workspace. A favorable essential efficiency value is also present, whereas the interface efficiency suffers from its necessity to complete a dialog for each created element.

MIT AppInventor Much like Alice and the other educational interfaces, MIT's AppInventor is designed for an unskilled, novice audience, reducing the breadth of target audience. AppInventor does not conform to the expected standards by only supporting three of the top ten popular features. While colorful, AppInventor's interface only supports four visual variables to distinguish elements in the chrome, relying heavily on icons and text. It combines the Alice style of visual syntax along with a Drag n Drop element creation workspace, resulting in high levels of effective and interface efficiency. Another artifact of the Alice style of visual syntax is its implicit syntax enforcement, assisting the user by

preventing illegal structures. Finally, the high level of language visual richness and modular complexity management scheme result in an overall visually pleasant experience.

MST Workshop Though MST Workshop offers a very simple and easy to use interface, it does not offer much explanation as to the use of its many different simulation categories. As such, they are unusable by anyone without prior knowledge of that subject, drastically limiting its accessibility. In the same vein, it does not incorporate enough visual richness variables to easily discern or interpret the tools and very few popular IDE features are supported. Though the efficiency values are above average and the clutter was rated to be relatively low, no actions were made to reduce the complexity of the system or even enforce the language's syntax. All of this merged together with a less-than-stellar language visual richness level shows that MST Workshop definitely has room for improvement.

Piet Creator As the primary IDE used in the creation of Piet programs, Piet Creator provides a very simple interface for novices without limiting the usability for more expert users. On top of that, the toolbars utilize seven of the eight tool visual richness variables and it possesses an extremely low clutter rating, maximizing accessibility for all users. However, Piet Creator does not offer any sort of properties dialog, eliminating the ability to manage data on a deeper level. The user is also limited to using only the mouse for every task, which creates a poor combination with the fact that Piet Creator holds the lowest interface efficiency value. The Piet language syntax is not enforced in the slightest, forcing the user to manually debug his whole program to locate any error in the code. In general, the simplicity of the design allows for a visually appealing environment, but creates a dearth of features.

Scratch Scratch is specifically targeted toward a novice audience with a low skill level, and only supports three of the top ten popular features. The interface employs an average level of visual richness variables, and does not even define an object properties window. The sparse, cluttered interface redeems itself through high values of essential and interface efficiency. Implicit syntax enforcement provides a safe environment for learning users, and the highest observed level of language visual richness provides an engaging, visually rich display. The focus on efficiency and visual richness works well in an IDE designed for education by accelerating reinforcement and engaging student attention.

Simulink Though Simulink features a plethora of components that can allow it to perform virtually any electrical simulation, the vast scope of its functions and the amount of on-screen tasks greatly reduces its accessibility. Each individual function that Simulink provides creates its own dialog window on screen, severely increasing visual clutter and complexity with prolonged use and no techniques for complexity management. A searchable toolspace is present to take off some mental load in dealing with Simulink's huge tool libraries, however very few visual richness variables are integrated to increase discernibility between the tools. The canvas elements also provide some relatively good efficiency values, though the IDE provides no features to enforce the syntax of the simulation language. The lack of visual richness overall detracts from the usability and enjoyment of this tool, despite it being very powerful.

Stencyl It provides a very easy-to-use interface for game software creation for an early level audience, but in the process sacrifices any higher level functionality. This IDE also integrates the Alice style of coding, which carries along with it very high efficiency values and implicit syntax enforcement. Every tool visual richness variable is utilized within the toolbars, which are also searchable, allowing for a very user-friendly and productive interface. Each language visual richness variable is supported within the canvas as well, thereby giving Stencyl perfect visual richness ratings. A very large number of perspectives are present due to Stencyl's powerful modularization techniques, greatly reducing the amount of mental strain on the user to handle the many steps that go into creating a game and showing Stencyl to be an altogether organized and well-designed IDE.

Tersus It is able to offer an interface for web application design with a very large number of popular IDE features, largely thanks to the integration of Eclipse's user interface. Unfortunately, very few visual richness variables are utilized for either the tools or the language and there is no syntax enforcement to aid the user at all through the design process. The efficiency of the IDE is at an acceptable level. Tersus hits a high point with its use of a hierarchical design to program behaviors and functions of the various components in the model. Though this is a very useful feature, it does not distract at all from the overly simplistic language.

TouchDevelop Designed as an educational program as part of Microsoft's "Hour of Code" campaign, TouchDevelop offers a simple user interface with a very limited functionality. Though the model itself is textually code based, the code is written by pressing buttons representing different commands, objects, attributes, etc., increasing both essential and interface efficiency by a good margin. However, the large, on-screen keyboard gives the interface a slightly cluttered feel and the text-based model places limits on the amount of language visual richness variables that can be integrated in the design.

UMLet UMLet provides a general purpose, easy-to-use software interface with no particular expectations about user skill level in Unified Modeling Language (UML). It offers many of the most popular IDE features a user might expect, but does not offer a visually rich chrome or language. Conceptually efficient but ultimately visually cluttered, UMLet measured high in essential efficiency but low in interface efficiency. To the detriment of the inexperienced user, UM-

Let offers no syntax enforcement. It also offers no complexity management devices to ease mental load. Barebones to the extreme, UMLet provides users with a visually simple interface and no additional usability features like context-sensitive tooling or searchable tools. Its power, however, lies in its simplicity and ease of use at any entry skill level.

Violet Similar to UMLet, Violet provides an easy-to-use interface with no specific entry skill level assumptions about UML. It offers the same common IDE features as UMLet, with a slightly richer chrome. The interface of Violet is slightly less cluttered than in UMLet, but also offers a lower essential efficiency. Violet is higher in interface efficiency over UMLet, though. Neither interface offers syntax enforcement or complexity management and, since both support UML as the target visual language, both share a low level of language visual richness. The two interfaces are relatively featureless and differ primarily in the mode of user interaction.

VisSim VisSim is a highly sophisticated visual simulation tool, designed for experts in the field. By supporting most popular IDE features, VisSim meets most developer expectations. The interface and language are adequately visually rich, supporting the average count of four visual variables. The large library of functionality is conveniently searchable, and the interface is highly efficient overall. VisSim has an easy-to-use hierarchical complexity management system which reduces overall mental load, but a lack of syntax enforcement may leave users in a dangerous position, forcing them to rely on verification at run-time only.

Visual Paradigm It offers a UML-based interface designed to support every defined UML diagram. As such, however, much of the functionality may be overwhelming to a novice to UML architecture. Many popular IDE features are present, though, as well as context sensitive tools for a slightly more accessible use. The design of Visual Paradigm allows for implicit syntax enforcement, a very useful feature when dealing with UML, with an acceptable interface efficiency and average essential efficiency value. An unfavorable amount of clutter is present in the IDE however, and no effort is taken to manage complexity throughout the interface. Visual Paradigm also suffers of an extremely low amount of language visual richness, due to UML standards.

Visual Use Case Unlike Visual Paradigm which supports the creation of all UML diagrams, Visual Use Case gives an in-depth approach to the requirements workflow with a focus on use case creation. Many different perspectives are provided as modularization tools to allow the micromanagement of the different use cases. Some perspectives utilize pieces of information from other perspectives for autocompletion, awarding Visual Use Case with a very high interface efficiency as well as the highest essential efficiency value. The extent at which the IDE goes into detail does not make it ideal for beginners, however, and very few popular IDE features are integrated in the design. Its high degree of visual richness stacks up rather poorly as well, and the syntax within the various perspectives is not enforced much. Overall, if Visual Use Case's users can get it past its lack of accessibility, it turns out to be a very powerful and in-depth tool for managing use cases.

WebRatio Similar to Tersus, WebRatio provides a web application creation interface with many popular IDE features. The overall visual richness of WebRatio is improved over that of Tersus, however the essential efficiency is slightly worse. WebRatio possesses a much larger amount of clutter as well, but it also incorporates a couple more perspectives for model management. Additionally similar to Tersus is the use of a hierarchical design to manage complexity within the system, although the syntax of the modeling language is not enforced either. Though WebRatio and Tersus tend to be similar in many ways, each have their own pros and cons which leaves it up for the user to determine his own preference of the two.

YAWL YAWL is a business workflow system that supports a simple, UML-like modeling language. Despite its simplicity, YAWL is targeted toward skilled members of its domain. YAWL is lacking in its support of popular IDE features and any form of active syntax checking. Complex diagrams are also difficult to manipulate given the lack of complexity management paradigms. YAWL also only supports a slightly lower than average number of visual richness variables in both the chrome and language. The language itself especially suffers from lack of visual richness with highly visually similar, square elements. However, YAWL's interface has a very low clutter value and high degrees of efficiency. Ultimately, the negatives outweigh the positives and, though minimal and efficient, YAWL is overall not visually appealing and poor in IDE features.

4. Discussion

Agreement on visual features indicates a common theme among studied IDEs . The feature that exhibited the most agreement between IDEs was the number of available perspectives, with 13 IDEs containing only a single one. The next most agreed upon features are interface visual richness and language visual richness which both share 9 IDEs that support only four visual richness variables for either interface or language. Note that IDEs do not necessarily employ the same number of visual richness variables for interface as well as supported language.

The prevalence of a single perspective in most studied IDEs indicates a tendency toward simplicity of user control. By introducing multiple perspectives, an IDE offers the user a richer set of controls and wider variety of views on the model, but also adds to the user's mental load in keeping track of new details.

As indicated by Moody [12] a larger magnitude of visual richness variables correlates to a more visually discrim-

inating and thus rapidly understandable interface. With most IDEs settling on four visual richness variables employed, the common theme among sampled IDEs is to a mid-range value. It is possible that too many visual richness features may be considered "junk", as per Tufte's warning [14].

Many IDEs would be able to benefit through the implementation of simple, positive features like the addition of visual richness variables. Additionally, only 11 IDEs implement context sensitive menus. Even simple convenience features, like the ability to search through available tools, are common in less than half of studied IDEs. This lack of simple, positive features actually tends to detract from the overall quality and usability of an interface.

Some IDE features can be related to one another. The collected data suggests that interface and essential efficiency are significantly related. A Pearson's product-moment correlation test indicates that the two variables share a correlation coefficient of 0.556 with $p = 0.003893$. Neither interface nor essential efficiency significantly correlate with visual clutter however, with $p > 0.4$ for each. The relationship between interface and essential efficiency is expected, not only due to similarity of the metrics use for each, but because of the underlying concepts. The more automation or efficiency an interface directly offers to its users, the more efficiency is provided to the user as a mental load deduction. That is, without the need to focus on details of implementation in the interface, the user is free to concern himself with other manually controlled details. These manually controlled details are emphasized as more important in the IDE designers by virtue of not being automated.

Overall, the relationships determined by this initial study indicate an emphasis on managing operator mental load in visual IDE design. Many sampled IDEs offer a number of visual variables to the user providing increased visual discrimination for ease of use. Additionally, the relationship found between interface and essential efficiency hints that interface design plays directly into the mangitude of user mental load.

5. Conclusion

The usability and suitability of an IDE can begin to be understood through an analysis of its interface characteristics. Design decisions involving the intended audience, the chrome or interface of the IDE, the style of human interaction, and features that affect level of language support all impact the overall usability and suitability. Even simple convenience features, like context sensitivity of searchable tools, can positively impact usability. Meanwhile, a lack of more integral characteristics, like visual richness, can very negatively impact the quality of an IDE.

The features discussed in this paper are inspired from software engineering approaches and methodologies. A more in-depth list of feature based on human-computer interaction theories will complement this study, such as the work by Green and Petre [7] of incorporating a cognitive dimension in visual programming environments. This is left for future work.

This is a preliminary study that is not exhaustive. Additional visual IDEs need to be considered to grow the body of collected data. Research into new features, revision of existing features, and additional analytical steps that follow after data collection would further the gained understanding of studied IDEs. The goal of this study has been to provide a technical foundation for the systematic analysis and understanding of domain-specific visual IDEs.

Acknowledgments

This research was sponsored by the National Science Foundation grant no. 1156563 at the University of Alabama REU site.

References

[1] D. V. Cicchetti. Guidelines, criteria, and rules of thumb for evaluating normed and standardized assessment instruments in psychology. *Psychological Assessment*, 6(4):284–290, 1994.

[2] L. Constantine. "Usage-centered software engineering: new models, methods, and metrics ". In *Software Engineering: Education and Practice, 1996. Proceedings. International Conference*, pages 2–9, Jan 1996. .

[3] A. Cooper, R. Reimann, and D. Cronin. *About face 3: the essentials of interaction design*. John Wiley & Sons, 2007.

[4] G. Costagliola, A. Delucia, S. Orefice, and G. Polese. A Classification Framework to Support the Design of Visual Languages. *Journal of Visual Languages & Computing*, 13 (6):573–600, 2002. .

[5] G. Fischer. Domain-oriented design environments. *Automated Software Engineering*, 1(2):177–203, 1994. .

[6] W. O. Galitz. *The essential guide to user interface design: an introduction to GUI design principles and techniques*. John Wiley & Sons, 2007.

[7] T. Green and M. Petre. Usability Analysis of Visual Programming Environments: A Cognitive DimensionsÒăFramework. *Journal of Visual Languages & Computing*, 7(2):131–174, jun 1996.

[8] A. Habermann and D. Notkin. Gandalf: Software development environments. *Software Engineering, IEEE Transactions on*, SE-12(12):1117–1127, Dec 1986. .

[9] K. A. Hallgren. Computing inter-rater reliability for observational data: An overview and tutorial. *Tutorials in quantitative methods for psychology*, 8(1):23, 2012.

[10] D. D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3(1):69–101, 1992.

[11] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008.

[12] D. Moody. The Physics of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, Nov 2009. .

[13] G. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *Software, IEEE*, 23(4):76–83, July 2006. .

[14] E. R. Tufte. *The visual display of quantitative information*. Graphics press Cheshire, CT, 2 edition, may 2001.

[15] Z.-G. Wei, A. P. Macwan, and P. A. Wieringa. A Quantitative Measure for Degree of Automation and Its Relation to System Performance and Mental Load. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 40(2):277–295, 1998. .