IDE Features and Comparison

Jean Michel Rouly Jonathan Orbeck

May 1, 2015

Contents

Foobar.

A IDE List

The following Integrated Development Environments (IDEs) were studied.

IDE	Version	Homepage
Alice3	3.1.92.0.0	http://alice.org
AToMPM	0.5.3	http://syriani.cs.ua.edu/atompm/atompm.htm
AudioMulch	2.2.4	http://audiomulch.com
Blender	2.71	http://blender.org
Cameleon	2.0.6	http://www.shinoe.org/cameleon/
EMF	2.9.2	http://www.eclipse.org/modeling/emf/
GRC	3.6.1	http://www.joshknows.com/grc
Grasshopper 3D	0.9.0075	http://grasshopper3d.com
Max	6.1	http://cycling74.com/products/max/
MetaEdit+	5.0	http://www.metacase.com/mep/
MIT AppInventor	nb135a	http://appinventor.mit.edu
MST Workshop	5.4.6	http://home.comcast.net/~tpandolfi/site/
Piet Creator	2011.4	https://github.com/Ramblurr/PietCreator/wiki
Scratch	2.0	http://scratch.mit.edu
Simulink	7.5.0	http://www.mathworks.com/products/simulink/
Stencyl	3.1.0	http://stencyl.com
Tersus	2.1	http://tersus.com
TouchDevelop	2014.6	https://touchdevelop.com
UMLet	12.2–1	http://umlet.com
Violet	2.0.1–1	http://alexdp.free.fr/violetumleditor
VisSim	8.0	http://vissim.com
Visual Paradigm	10.2	http://visual-paradigm.com
Visual Use Case	7.22	http://visualusecase.com
WebRatio	7.2.4	http://webratio.com
YAWL	3.0	http://yawlfoundation.org

Table 1: IDE list

B Variable Definitions

The variable definition list will contain at minimum five common features for each variable. These will include:

name the full, unique name of the variable

abbreviation a shortened, unique identifier

type variable data type, one of *boolean*, *nominal*, *ordinal*, *ratio*, or *compound*

description a detailed description of the variable and any references

scoring a description of how the variable affects the overall performance and usability of the IDE

Variable definitions may also include **accepted values** or, occasionally, **component variables**.

B.1 Audience

The **Audience** of an IDE refers generally to the target populations for which the IDE is developed.

B.1.1 Domain

Abbreviation `Domain`

Variable Type Nominal

Description The **Domain** of an IDE refers to the *domain* or field of knowledge under which the interface falls. That is, the field or fields for which the interface is primarily used, if any. Example values might include `General`, `Modeling`, `Software`, *etc.*

Accepted Values

General	This IDE is general purpose. It can be applied to or used within any number of different fields to the same effect.
...	Any custom field is allowed.

Scoring This variable is neutral. Domain is determined simply by the purpose of the IDE and has no meaningful impact on usability.

B.1.2 Skill Level

Abbreviation `Skill`

Variable Type Nominal

Description The **Skill Level** of an IDE describes what skill level within the given **Domain** is expected of its users.

Accepted Values

Novice	The user is expected to have little to no experience in the domain. The IDE is designed to be user friendly and welcoming to new users with no former domain knowledge.
Intermediate	The user is expected to have a medium level of experience in the domain. The IDE may offer some advanced functionality, but still be welcoming to newer users.

	Expert	The user is expected to have a high level of experience in the domain. The IDE offers highly sophisticated functionality that the user is expected to understand without further instruction.
	General	The IDE offers a powerful set of advanced functionality while maintaining an overall level of accessibility and ease of use.

Scoring This variable can score positively or negatively. The value `General` is ideal and scores positively because it indicates a high level of accessibility balanced with a similar level of utility. Any other value indicates specialization and, while perhaps increasing utility, decreases usability overall, thus scoring negatively.

B.2 Chrome

The **Chrome** of an IDE is the sum total of user interface components external to the workspace. This includes every tool, menu, button, or other user interface component not contained within the active workspace area.

B.2.1 Popular Features

Abbreviation `Features`

Variable Type Compound

Description Many IDEs provide support for a set of common operations. [6] defines a list of the top 10 most frequently used IDE features, which are reproduced here as boolean sub-variables. Each sub-variable indicates whether the IDE in question supports the use of that particular feature.

Components	**Delete**	Delete a syntactic element in the workspace
	Save	Save and export a model to a storage media
	Paste	Duplicate a syntactic element in the workspace from the paste buffer
	Content Assist	Provide suggestions or completion for elements
	Copy	Place syntactic element from the workspace into paste buffer
	Undo	Undo the user's most recent action
	Cut	Place a syntactic element from the workspace into paste buffer, and remove from workspace
	Refresh	Load contents of workspace and interface dashboard elements from storage media and update display if necessary
	Show View	Open and display a new tool in the interface
	Next Word	Move active selection to the next element according to some natural ordering

Scoring This variable scores positively with a larger total count of present subvariables. With more popular features present, an IDE accessibility through conformity with common user expectations.

B.2.2 Context Sensitive Tools

Abbreviation `Context`

Variable Type Boolean

Description Any interface component which changes visibly or is generated anew depending on the context of selected elements within the workspace is a **Context Sensitive Tool**.

Accepted Values

Yes	The IDE supports context sensitive tools.
No	The IDE does not support context sensitive tools.

Scoring This variable scores positively if present. Context-sensitive tools are ubiquitous in graphical user interfaces and provide a simple yet conventional way to increase usability.

B.2.3 Degree of Interface Visual Richness

Abbreviation `ToolRichness`

Variable Type Compound

Description The **Degree of Interface Visual Richness** describes the extent to which an interface utilizes visual variables to increase visual discriminability of available tools. The component variables measured are a composite of several authors' research, including elements from visual language design. Each component variable is a boolean measure determining whether or not the described visual variable is used in the interface to distinguish available tools.

Components

Icons	Images contained in a border of a standard size and shape represent distinct actions or tools. [3, 5]
Shape	Distinct shapes indicate different tools or classes of tool. [5]
Size	Different tool sizes indicate distinct tools or classes of tool. [5]
Color	Color is used to indicate distinct tools or classes of tool. [5]
Text	Text (or typographic variation) is used to identify or distinguish tools or classes of tool. [5]
Organizational Coherence	Components with related purpose are visually grouped in the interface. [2]
Texture	Shading or shadows are used to modify tools or to distinguish between distinct tools or classes of tool. [5]
Brightness	The brightness of a color (*i.e.* its perceived luminosity) is used to indicate a difference between tools or classes of tool. [5]

Scoring This variable scores positively, increasing with the number of present component variables. A higher level of visual discriminability simplifies the task of differentiating elements on the screen for the user.

B.2.4 Multiplicity of Perspectives

Abbreviation `Perspectives`

Variable Type Ratio

Description The number of available predefined interface perspectives available to the user. A perspective is defined as a visual configuration of the available tools in the interface chrome and elements in the workspace for the purposes of accomplishing a distinct task by means of a distinct process.

Range $[1, \infty)$

Scoring This variable scores positively with a direct correlation to the number of perspectives supported. Different perspectives are able to modularize complex systems and reflect data in various visual manners, simplifying use of the IDE and adhering to the preferences of a wide range of users.

B.2.5 Object Properties Window

Abbreviation `Properties`

Variable Type Nominal

Description The **Object Properties Window** is any interface component that displays the properties of an element in the workspace. This component generally allows modification of element properties as well.

Accepted Values

Omnipresent	A single property dialog or window is always present. The contents may update contextually.
Manual	The system requires user interaction to present the properties window.
None	No properties window is ever presented for workspace elements.

Scoring This variable is neutral as long as it is present, whereas a lack of a properties dialog would be scored negatively. Properties dialogs offer the user a means to alter in-depth information about an element or a group of elements and the absence of one eliminates this level of customizability.

B.2.6 Searchable Toolspace

Abbreviation `Searchable`

Variable Type Boolean

Description The *toolspace* is the total set of available tools, components, or actions that the IDE offers to the user. If an IDE employs a **Searchable Toolspace**, it allows the user to search through this toolspace by name or keywords.

Accepted Values

Yes	The IDE supports a searchable toolspace.
No	The IDE does not support a searchable toolspace.

Scoring This variable is scored positively if present. The ability to search for a tool by name, especially one whose location is unknown by the user, drastically reduces the time and mouse clicks required to manually search through menus to find the desired tool.

B.2.7 Toolbar Styles

Abbreviation `ToolStyle`

Variable Type

Description

Accepted Values ...

Scoring This variable is neutral. The usefulness of each **Toolbar Style** is dependent on the function of the IDE and the purpose of each individual toolbar.

B.2.8 Visual Clutter

Abbreviation `Clutter`

Variable Type Ordinal

Description **Visual clutter** is defined as the number and organization of tools available on the screen versus the amount of workspace provided by the IDE. 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 visually cluttered.

Accepted Values

Low	There is a minimal and cohesive amount of tools present on the screen.
Medium	There is a somewhat organized and manageable amount of tools available on the screen.
High	There is an unorganized or overwhelming amount of tools taking up space on the screen.

Scoring This variable is scored negatively with an increasing amount of **visual clutter**. A large amount of clutter results in an interface that is difficult to navigate, places unnecessary strain on the user's mind, and decreases the overall intuitiveness of the IDE.

B.3 Human Interface

Human Interface components of an IDE include aspects of the software interface that affect how the human 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).

B.3.1 Essential Efficiency

Abbreviation `EEfficiency`

Variable Type Ratio

Description 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 to the number of steps in the representative essential use case (see Appendix D). [2]

$$1 - \frac{\text{Number of steps in concrete use case}}{\text{Number of steps in essential use case}}$$

Unlike the metrics for automation put forth in [7], this measure does not require user experience reports, and can be measured through simple use case analysis.

Range $[0, 1]$

Critical Values

1	There are no steps required to complete the concrete use case. Not feasible in practice.
0	Indicates a one-to-one relationship between the number of steps to complete a concrete use case and the number of steps in the essential use case.

Scoring This variable is scored positively based on its value. The more **Essential Efficiency** an interface possesses, the less mental work the user needs to perform.

B.3.2 Interface Efficiency

Abbreviation `IEfficiency`

Variable Type Ratio

Description 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) required of the operator to complete a task compared against the number of abstract steps in the representative essential use case:

$$1 - \frac{\text{Number of physical actions to complete use case}}{\text{Number of steps in essential use case}}$$

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.

Range $(-\infty, 1)$

Critical Values

1	There are no physical actions required. Complete automation, not feasible in practice.
0	Indicates a one-to-one relationship between the number of physical actions to complete a use case and the number of steps in the essential use case.
< 0	Indicates a larger number of physical actions than steps in the essential use case. The lower the value, the less the degree of automation provided by the interface.

Scoring This variable is scored positively based on its value. The more **Interface Efficiency** an interface possesses, the less physical work the user needs to perform.

B.3.3 Keyboard Use

Abbreviation `Keyboard`

Variable Type Ordinal

Description **Keyboard Use** 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.

Accepted Values

None	The IDE does not offer the use of a keyboard.

	Simple	The keyboard is used only for typing annotations, properties, or comments.
	Optional	The option to use a keyboard for some actions is available, but these actions can also be performed with a mouse.
	Required	There are some actions which can be only be performed with a keyboard.
Scoring	Each value for this variable is scored differently. `Simple` is neutral, `optional` is positive, and `required` and `none` are negative. Optional keyboard increases user ability to interact with the IDE, allowing the user to determine their optimal mouse-and-keyboard combination. Required keyboard use, on the other hand, locks the user into a possibly uncomfortable or unintuitive method of interaction (similarly with supporting only mouse use).	

B.3.4 Mode of Element Creation

Abbreviation	`Mode`	
Variable Type	Nominal	
Description	The process by which the user creates elements in the given 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. The *Point n Click* process utilizes a single mouse click to indicate a selection and followed with subsequent mouse clicks elsewhere to define placement.	
Accepted Values	**Drag n Drop (1:1)**	A Drag n Drop method is used to create elements. At most one element can be created per action.
	Drag n Drop (1:n)	A Drag n Drop method is used to create elements. Multiple elements can possibly be created per action.
	Point n Click (1:1)	A Pint n Click method is used to create elements. At most one element can be created per action.
	Point n Click (1:n)	A Point n Click method is used to create elements. Multiple elements can possibly be created per action.
Scoring	This variable is neutral. Although `Point n Click (1:n)` is slightly more productive than the other two possible values, the difference is minuscule enough to be deemed insignificant.	

B.3.5 Tertiary Interface Devices

Abbreviation	`Devices`	
Variable Type	Nominal	
Description	**Tertiary Interface Devices** refers to any third-party human interface devices which can be used to interact with the interface. This could include music alteration devices such as microphones and MIDI keyboards as well as the integration of mobile devices.	
Accepted Values	**None**	The IDE does not offer any functionality for third-party interface devices.
	...	Any custom field is allowed.
Scoring	This variable is neutral. Though being able to interface with multiple devices might be beneficial at times, depending on the domain it might not even be feasible.	

B.4 Integration

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.

B.4.1 Allowed Relations Indicated

Abbreviation `Relations`

Variable Type Boolean

Description 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.

Accepted Values

Yes	The IDE emphasizes the allowed relations.
No	The IDE does not emphasize the allowed relations.

Scoring This variable is scored positively if present. While not having this feature is not necessarily a bad thing, its presence can reduce mental load on the user by quickly indicating what actions are allowed to them.

B.4.2 Output Generation Style

Abbreviation `Output`

Variable Type Nominal

Description **Output Generation Style** describes the mode with which the IDE renders and displays output to the user. It is a dual axis 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.* a model). *Live/trigger* describes whether output is generated and displayed live or after some event triggered by the user.

Accepted Values

Direct Live	Any modifications directly alter the output, which is dynamically updated.
Indirect Live	Any modifications indirectly alter the output (*e.g.* via a model), which is dynamically updated.
Direct Trigger	Any modifications directly alter the output, but a final compilation or execution process is required to complete the output.
Indirect Trigger	Any modifications indirectly alter the output (*e.g.* via a model), but a final compilation or execution process is required to complete the output.

Scoring This variable is neutral and is more indicative of an IDE's purpose than its quality. No combination of values (`direct`, `indirect`, `dynamic`, and `trigger`) results in a greatly improved or reduced quality of user experience.

B.4.3 Syntax Enforcement

Abbreviation `Syntax`

Variable Type Nominal

Description **Syntax Enforcement** measures the mode with which the IDE enforces language syntax requirements.

Accepted Values

Explicit	The IDE displays a message when the user creates a visual syntax error.
Implicit	The IDE incorporates features that prevent the user from creating visual syntax errors.
None	The IDE does not enforce visual syntax, or it is only enforced when explicitly checked (*e.g.* compiled, executed, *etc.*).

Scoring This variable can be scored positively or negatively, depending on the values it takes. `Implicit` is scored highest, followed by `explicit`, followed by `none`. Implicit syntax enforcement reduces the amount of load placed on the user by reducing the number of available actions to a small set of legal actions. Explicit enforcement informs the user of their mistakes after the fact and requires user action to correct them. The lack of any syntax enforcement is dangerous and does not provide the user any assistance.

B.5 Language Syntax

The **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 document.

B.5.1 Complexity Management

Abbreviation `Complexity`

Variable Type Nominal

Description 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. [5]

Accepted Values

Modularization	Larger systems within the language are divided into smaller subtasks. [5]
Hierarchy	Different systems within the language are represented at different levels of detail. [5]
None	The language makes no effort to separate components.

Scoring This variable is scored positively if present and negatively otherwise. `Modularization` and a `hierarchy` are both effective means of managing complexity, with neither objectively better than the other. However, a lack of complexity management can lead to a confusing or overwhelming design, particularly in interfaces which incorporate larger systems.

B.5.2 Connection Style

Abbreviation `Connection`

Variable Type Nominal

Description A language's **Connection Style** refers to the manner that element connections are displayed. This dual axis variable measures connections as *overlapping/linked* as well as connection sources as *point/region* based. The former axis describes the visual representation of connections, while the latter describes how links are connected. If the supported language does not register on these axes, it is not "Connection-based" [3] and must be *geometric*.

Accepted Values

Overlapping Points	The elements are connected by positioning specific points to overlap. [3]
Overlapping Regions	The elements are connected by positioning specific regions to overlap. [3]
Linked Points	The elements are connected by linking together distinct points on each element. [3]
Linked Regions	The elements are connected by linking together distinct regions on each element. [3]
Geometric	The elements are connected via geometric placement. [3]

Scoring This variable is neutral. The **connection style** is more dependent on the visual language being used and is not a good representation of the IDE's quality or features.

B.5.3 Degree of Language Visual Richness

Abbreviation `LanguageRichness`

Variable Type Compound

Description

Accepted Values

Icons	Images contained in the elements are used to represent different elements. [3, 5]
Shape	Distinct shapes indicate different elements or classes of element. [5]
Size	Different element sizes indicate distinct elements or classes of element. [5]
Color	Color is used to indicate distinct elements or classes of element. [5]
Text	Text (or typographic variation) is used to identify or distinguish elements or classes of element. [5]
Texture	Shading or shadows are used to modify elements or to distinguish between distinct elements or classes of element. [5]
Brightness	The brightness of a color (*i.e.* its perceived luminosity) is used to indicate a difference between elements or classes of element. [5]

Orientation Rotation of an element indicates differences between distinct elements or classes of element. [5]

Horizontal Position The horizontal location of an element indicates differences between elements. [5]

Vertical Position The vertical location of an element indicates differences between elements. [5]

Scoring This variable scores positively, increasing with the number of present component variables. As with tool richness, a higher level of visual discriminability simplifies the task of differentiating elements on the screen for the user.

C Interface and Essential Efficiency

	Use Case	Physical Actions	Concrete Steps
Alice3	Open File (D.1.1)	7	3
	Create Element (D.1.2)	6	3
	Create and Link Elements (D.2.1)	12	7
AToMPM	Open File (D.1.1)	3	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	11	10
AudioMulch	Open File (D.1.1)	3	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	9	9
Blender	Open File (D.1.1)	4	3
	Create Element (D.1.2)	6	4
	Create and Transform Elements (D.2.3)	12	10
Cameleon	Open File (D.1.1)	4	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	9	10
EMF	Open File (D.1.1)	3	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	5	10
GRC	Open File (D.1.1)	3	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	5	8
Grasshopper 3D	Open File (D.1.1)	4	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	9	9
Max	Open File (D.1.1)	4	3
	Create Element (D.1.2)	4	4
	Create and Link Elements (D.2.1)	11	11
MetaEdit+	Open File (D.1.1)	6	3
	Create Element (D.1.2)	5	3
	Create and Link Elements (D.2.1)	14	9

	Use Case	Physical Actions	Concrete Steps
AppInventor	Open File (D.1.1)	3	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	6	7
MST Workshop	Open File (D.1.1)	4	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	8	8
Piet Creator	Open File (D.1.1)	3	3
	Create Element (D.1.2)	3	3
	Print an Integer (D.2.2)	16	10
Scratch	Open File (D.1.1)	5	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	6	7
Simulink	Open File (D.1.1)	3	3
	Create Element (D.1.2)	4	3
	Create and Link Elements (D.2.1)	11	9
Stencyl	Open File (D.1.1)	4	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	6	7
Tersus	Open File (D.1.1)	4	3
	Create Element (D.1.2)	4	3
	Create and Link Elements (D.2.1)	11	9
TouchDevelop	Open File (D.1.1)	3	3
	Create Element (D.1.2)	3	2
	Print a Character (D.2.4)	6	5
UMLet	Open File (D.1.1)	4	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	15	9
Violet	Open File (D.1.1)	4	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	11	11
VisSim	Open File (D.1.1)	3	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	9	9

	Use Case	Physical Actions	Concrete Steps
VisualParadigm	Open File (D.1.1)	4	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	10	10
Visual Use Case	Open File (D.1.1)	4	3
	Create Element (D.1.2)	4	3
	Create and Link Elements (D.2.1)	8	6
WebRatio	Open File (D.1.1)	4	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	10	10
YAWL	Open File (D.1.1)	3	3
	Create Element (D.1.2)	3	3
	Create and Link Elements (D.2.1)	9	9

D Essential Use Cases

An **Essential Use Case** is a simple, brief, technology-independent description of the steps necessary to a system use case. [2] Below, several example use cases are enumerated. The first group of use cases (D.1) are considered simple use cases, and are common to every IDE. The remaining use cases (D.2) are more complex or specialized, and may not be suitable for more than a few IDEs.

D.1 Simple

D.1.1 Open File

Opening a file is a simple, fundamental Essential Use Case for any IDE, visual or otherwise. It refers to fetching a single file from storage and loading it into the active editing workspace.

1. Indicate that you want to open a file.
2. Indicate which file you want to open.
3. Open the file.

D.1.2 Create Element

Visual languages are primarily composed of networks of linked elements. Therefore, creating one such element is a fundamental Essential Use Case. This Use Case requires that the element must be created and placed at a specific, user defined location.

1. Indicate that you want to create an element.
2. Indicate which element you want to create.
3. Designate the position at which you want to create the element.
4. Create the element.

D.2 Complex

D.2.1 Create and Link Elements

Visual Languages also allow elements to share links to one another. This Use Case requires two unique elements to be created and placed by the user, and then linked together in a meaningful fashion.

1. Indicate that you want to create an element
2. Indicate which element you want to create
3. Designate the position at which you want to create the element
4. Create the element
5. Indicate that you want to create another element
6. Indicate which element you want to create
7. Designate the position at which you want to create the element
8. Create the element
9. Indicate that you want to begin linking elements

10. Indicate type of connection to create.
11. Designate the first element to be linked
12. Designate the second element to be linked
13. Create the link

D.2.2 Print an Integer

1. Indicate which color will be the starting color.
2. Create the color block of size x.
3. Determine which color represents “push”.
4. Indicate that you will begin to use that color.
5. Create its color block.
6. Determine which color represents “out (integer)”.
7. Indicate that you will begin to use that color.
8. Create its color block.
9. Determine where any black codels will need to be.
10. Select the color black.
11. Create the black codels.
12. Execute the program.

D.2.3 Create Element and Transform

1. Indicate that you want to create an element
2. Indicate which element you want to create
3. Designate the position at which you want to create the element
4. Create the element
5. Indicate that you want to edit the element
6. Indicate that you want to transform a vertex
7. Select a vertex to transform
8. Indicate the type of transformation to perform
9. Perform transformation
10. Commit changes to element

D.2.4 Print a Character

1. Designate the position at which you want to create an element.
2. Indicate that you want to create a character.
3. Designate the character that you want to create.
4. Create the character.
5. Designate the position at which you want to place “post to wall”.
6. Indicate that you want to create a new command.
7. Indicate that you want to create “post to wall”.
8. Create the command.
9. Run the program.

E Methods

E.1 Mechanical Turk

We took three unique screenshots of each IDE and darkened the workspace area to remove attention from complex models rather than complex IDE chrome. We then distributed these screenshots on Amazon.com Mechanical Turk (MTurk) Human Intelligence Tasks (HITs), requiring five unique workers for each screenshot.

We ended up with 12 unique workers completing the tasks, identified anonymously in Table 3. On average, each worker spent about one minute studying an image of the IDE and received $0.02 for each task completed for an effective hourly rate of $1.00.

Worker Identifier	**HIT Count**
Worker A	21
Worker B	50
Worker C	59
Worker D	10
Worker E	5
Worker F	75
Worker G	4
Worker H	19
Worker I	56
Worker J	14
Worker K	61
Worker L	1

Table 3: Amazon MTurk workers

We transform the resulting data into a set of five vectors, represented similarly to the abbreviated Table 4 where numeric values are ordinal data representing the variable **Visual Clutter** on a scale from one to five, from low to high amounts of clutter.

IDE	**raterA**	**raterB**	**raterC**	**raterD**	**raterE**
alice1	3	2	5	4	4
alice2	4	2	4	4	5
alice3	2	5	4	1	2
appinventor1	2	5	4	5	4
appinventor2	1	3	3	4	3
appinventor3	4	3	3	3	2

Table 4: Amazon MTurk data preview

Because multiple reviewers were presented with each subject, 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, 4] 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 participating reviewers total, despite five sets of reviews (*i.e.* the experimental design is not "fully crossed" [4]). We argue that this is not significant, however, because the five sets of reviewers are disjoint sets, acting as entirely independent actors.

F Results

Alice3 Designed primarily as an educational programming environment, Alice3 alienates skilled users by expecting a lower 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 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 "MIT Scratch" style of visual syntax. However, despite greatly reducing operator mental load, Alice3's clunky 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.

AToMPM AToMPM is unique among the IDEs in this study in that is built as a modeling framework specifically to develop 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 the 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. It supports a large amount of popular IDE features, however, 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 or provides 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, its featurefulness leads directly to the second highest observed value for visual clutter. Blender possess 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.

Cameleon Intended for rapid, flexible visual prototyping of functional algorithms, Cameleon is relatively feature impoverished — it only supports three of the most popular IDE features. It does however boast a large number of available tools which are conveniently searchable. Cameleon's interface chrome employs five visual richness variables, slightly above the average count. Its sleek, simple interface is rated as low clutter, and provides a positive level of essential and interface efficiency. Interaction requires use of the keyboard for non-essential actions, specifically advanced navigation and zooming. Cameleon 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, Cameleon is an efficient, simple tool with a large library of functionality and a focus on supporting syntax requirements.

Eclipse Modeling Framework The 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 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 GNU Radio Companion 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. GRC is not extensively visually pleasing, however, as the interface chrome 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 some optional keyboard use and explicit syntax checking. GRC is designed as an easy to use interface for non-coders, 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. Complexity management is not supported at all, however, and visual richness in both the language and tools are mediocre at best.

Max Max's overly simple appearance can be misleading — it is very much designed for skilled users in the music development domain. It supports a great deal of popular features and an average number of visual richness variables in both its chrome and supported language, making it accessible and relatively visually pleasing. 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 Max's 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+ 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 daunting for newer users. These tasks are fortunately divided through modularization, allowing them to be much more manageable withing 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 MetaEdit+'s necessity to complete a dialog for each created element.

MIT AppInventor Much like Alice3 and the other educational interfaces, MIT's AppInventor is designed for an unskilled, novice audience, reducing the breadth of target audience. By only supporting three of the top ten popular features, AppInventor additionally alienates its audience through nonconformity with expected standards. While colorful, AppInventor's interface only supports four visual variables to distinguish elements in the chrome, relying heavily on icons and text. AppInventor combines the "MIT Scratch" 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 "MIT Scratch" 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 virtually 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 better than decent 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 does a very good job of providing 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/her whole program to located any errors in the code. In general, the simplicity of Piet Creator's design allows for a relaxed and visually appealing environment, but it can evidently be overly threadbare to a degrading extent.

Scratch One of the earliest educational interfaces present in this study, Scratch has influenced a great deal of later IDEs, including Stencyl, AppInventor, and Alice3 to name a few. 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 Stencyl 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 "MIT Scratch" 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 Tersus is able 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 decent, however, and 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 nice feature, it doesn't distract at all from the overall blandness of the 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. The large, on-screen "keyboard" gives the interface a slightly cluttered feel, however, 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. 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 clunky in implementation, UMLet measured high in essential efficiency but low in interface efficiency. To the detriment of the inexperienced user, UMLet 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. It offers the same common IDE features as UMLet, with a slightly richer chrome. Violet's interface is slightly less cluttered than UMLet's 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 acuml 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 overall the interface is highly efficient. VisSim's simple to use hierarchical complexity management system reduces overall mental load, but a lack of syntax enforcement may leave users in a dangerous position.

Visual Paradigm Visual Paradigm offers a acuml based interface designed to support every defined acuml diagram. As such, however, much of the functionality may be daunting to a novice to acuml architecture. Many popular IDE features are present, though, as well as context sensitive tools for a slightly more accessible

use. Visual Paradigm's design allows for implicit syntax enforcement, a very useful feature when dealing with acuml, with a decent amount of interface efficiency and an okay 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. Combined with an extremely low amount of language visual richness, due to acuml standards, the downsides of Visual Paradigm outweigh the good.

Visual Use Case Unlike Visual Paradigm which supports the creation of all Unified Modeling Language (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. It's all around 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/her own preference of the two.

YAWL YAWL is a workflow system that supports a simple, acuml-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 a visually boring, feature impoverished IDE.

References

[1] Domenic V. Cicchetti. Guidelines, criteria, and rules of thumb for evaluating normed and standardized assessment instruments in psychology. *Psychological Assessment*, 6(4):284 – 290, 1994. ISSN 1040-3590. URL `http://search.ebscohost.com/login.aspx?direct=true&db=pdh&AN=1995-15835-001&site=ehost-live`.

[2] L.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. doi: 10.1109/SEEP.1996.533974.

[3] 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. ISSN 1045-926X. doi: http://dx.doi.org/10.1006/jvlc.2002.0234. URL `http://www.sciencedirect.com/science/article/pii/S1045926X0290234X`.

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

[5] D.L. Moody. The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. *Software Engineering, IEEE Transactions on*, 35(6):756–779, Nov 2009. ISSN 0098-5589. doi: 10.1109/TSE.2009.67.

[6] G.C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the Eclipse IDE? *Software, IEEE*, 23(4):76–83, July 2006. ISSN 0740-7459. doi: 10.1109/MS.2006.105.

[7] Zhi-Gang Wei, Anil P. Macwan, and Peter 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. doi: 10.1518/001872098779480406. URL `http://hfs.sagepub.com/content/40/2/277.abstract`.