

Blank: An Insight-Based Approach To Application Development

Paul Go

February 11, 2018

Abstract

High quality mobile apps take months or years to build. Iterations take weeks. App development costs range from tens of thousands of dollars, to millions of dollars in some cases. The seriousness of this problem extends well beyond the obvious cost issues. A slow and expensive software development process is starkly misaligned with how business-minded app creators achieve product/market fit: constant iteration and reevaluation, with directional changes occurring sometimes on a weekly basis. Directional changes can inflate the total project cost, impair the delivery schedule, and lead to project failure.

In a sense, the high-cost / low-speed nature of app development has caused the industry to resemble an oligarchical dominance hierarchy. Today, the current economics do not allow software to come to fruition unless the high costs can be justified and supported.

We propose that these flaws are a consequence of app development not operating at the same level of abstraction that the individual who desires its creation is thinking. Stated another way, even the most modern programming languages such as TypeScript, Swift or Kotlin are still largely a representation of the inner processes of a computer, rather than the high level end-goals of the product.

This paper introduces a new high-level software development paradigm, *Insight-Oriented Programming* (henceforth referred to as “IOP”). IOP allows non-programmers to “program” by producing structured software specifications, which are concise 1-2KB quasi-natural language documents. These specifications are delivered to an automated reasoning compiler which produces a complete mobile app with a high quality user experience. These apps are delivered directly to the incumbent app stores, or can be segmented into small modular pieces and transplanted in other apps. Consumption of these embeddable pieces is governed through a crypto token, allowing for beneficial data sharing monetary agreements between disparate apps.

IOP achieves such feats through constructs that would be challenging to represent in a traditional text-based editing environment effectively. This paper therefore specifies a *Structured Editor* in which IOP documents are created.

Consider an ecosystem where apps are created by laypeople, for near-zero cost, are downloaded by consuming a negligible amount of data, installed instantaneously, provide a superior user experience, and exhibit an uncanny ability to integrate and blend with each other. Such an ecosystem would raise questions about whether the public internet itself would remain useful. This is the end-goal of Blank.

Contents

1	Insight Oriented Programming	3
1.1	Introduction	3
1.2	Scope	3
1.3	Formalizing The Specification	3
1.4	Relationship To Traditional Programming	4
1.5	Typing Discipline	5
1.5.1	Taxonomic Typing	5
1.5.2	Predictive Typing	5
1.6	Document Structure	6
1.7	Labels	8
1.8	Settings	9
1.9	Tables	9
1.10	Fields	10
1.11	Procedures	10
1.12	Formulas	11
1.12.1	Pronouns	12
1.12.2	Operators	12
1.12.3	Literals	14
1.13	Complete Example	14
2	Automated User Interface Generation	19
2.1	Scope	19
2.2	Inferring Navigational Structure From Data Relationships	19
2.2.1	Basic Algorithm	20
2.2.2	Screen Types	20
2.2.3	An Abstract Representation Of Navigation	22
2.2.4	Real-World Examples	23
2.3	Deterministic Screen Layout Optimization	24
2.3.1	List Screen Layout	24
2.3.2	Object Screen Layout	25
2.3.3	Preview Layout	26
3	Data Management	28
3.1	Microservice Generation	28
3.2	Distributability	28
4	Public Interfaces	29
4.1	Formats	29
4.2	Polymorphism	30
5	FAQ	31
6	Conclusion	31

1 Insight Oriented Programming

1.1 Introduction

Historically, “programming” has come to refer to the process of defining a logical system that coordinates the flow and transformation of input data, potentially generating output. IOP was built upon the finding that within the context of app development, if a compiler is supplied with sufficient insight into the semantic nature of the data, the flow and transformation of that data can be predicted, and therefore implemented automatically.

IOP appears to be the first method of achieving general purpose programming of production apps, where all computations performed are done so in a non-Turing-complete manner. Avoiding Turing-completeness was a design goal, as it dramatically improves comprehensibility by avoiding potentially confusing concepts such as looping and recursion.

But IOP goes further. It is known that a user interface is a projection of a data model. However, when the insights pertaining to how a user will interact with that data model are limited, the possibility of generating a user interface automatically is conversely limited to highly generic use cases such as internal administrative software¹. Such is not the case in the context of IOP, as the available insight can be used to produce optimized and highly specific user experiences, meeting the needs of the majority of consumer-grade apps, even when world-class quality and widespread adoption is a goal.

IOP brings the capacity for app development to a far greater number of people, dramatically reduces cost, and lays the technical groundwork for apps to be downloaded, installed, and launched on a phone at near-instant speed (solutions to the obvious real-world implementation challenges with respect to the app stores are covered in this paper).

1.2 Scope

IOP is intended to expedite the production of most apps that are being built on handheld mobile devices. More specifically, IOP is targeted towards apps that are information-centric in nature, such that their primary focus is the collection, sharing, and display of information. This of course includes a very wide spectrum of apps in verticals ranging from marketplaces, line-of-business, shopping, sharing, lifestyle, entertainment, news, and more.

IOP is not appropriate where the highly specialized user interface is a hard requirement. Examples include CAD modeling, games, and augmented reality.

Future work will expand the system to support tablets and potentially laptops and desktops.

1.3 Formalizing The Specification

The scenario is all too common. An individual or organization has a need for an app to address a particular business need, so a document listing a set of requirements is assembled. Lacking the technical skills to construct the app themselves, they commission a development organization who specializes in the process to handle the implementation. However, after the initial meeting, it becomes clear that their understanding of their need is not as fully formed as they thought, and therefore, are not ready to enter a build phase. Through a series of meetings, the development organization helps the client produce a detailed specification so that implementation can begin.

¹An example of such software can be found at <https://trestle.io>

The build phase then begins, but as updates are presented, the client becomes disappointed due to a combination of factors such as:

1. The development company discovering small edge cases that are not addressed in the specification, and handling them in ways that the client views as unsatisfactory.
2. The client having a different interpretation of the specification than the development company.
3. The development company taking shortcuts that still achieve conformance with the specification, but lead to a less than optimal user experience.
4. The client uncovering new information about their need, and wanting to make changes to the specification, but presenting these not as changes, but as a conveniently alternative interpretation in hopes of coercing the development company into providing additional free labour.

Such complications arise because requirements and specifications are assembled in an unstructured format that allows for interpretation, and offer no way to systematically verify the completeness. IOP offers an automatic alternative to these processes. Its novel design forces the important questions to be asked, such as whether users can run a system-wide search for objects of a particular type, or whether certain objects can be deleted or renamed.

1.4 Relationship To Traditional Programming

Various programming languages enable analysis tools (such as compilers, interpreters, editing environments, and documentation generators) to extract varying amounts of insight from the code. “Insight” in this context refers to the degree to which an analysis tool can understand what a given piece of code will and will not do. Although some languages place the ability to gather insight at a higher level importance than others, supplying insight is a desirable quality in any language. The more insight a tool can draw from a particular block of code, the more clearly that tool can reason about it’s behavior, and therefore, the more powerful the tooling surrounding the language can become. This is true of any programming system, whether text-based (Java), or interface-based (Scratch).

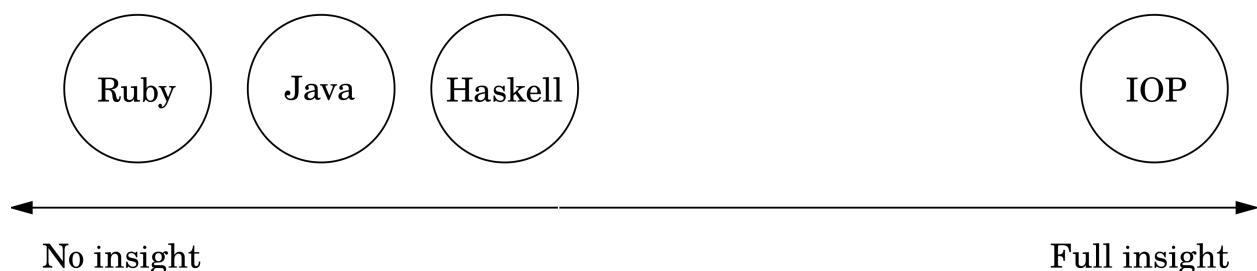


Figure 1: An example of some programming languages as they would fall on a spectrum of the degree to which a language provides insight to the host environment.

Ruby provides very little insight to the surrounding environment. As a result, execution is limited to performance-impaired interpreters, and common editing functions like rich statement completion and referential navigation are unavailable.

Java offers a rudimentary type system, which makes possible modern capabilities like compilation, reasonably rich editing, and various static analysis tools.

Haskell is heavily disposed toward stateless functional programming, isolating all state via monadic structures. As a result, the Haskell compiler has the freedom to be significantly more aggressive with performance optimizations than typical languages, oftentimes generating executable code that uses a radically different structure than the uncompiled counterpart.

IOP is a departure from traditional notions of programming all together, to the point where it raises questions about whether it is valid to classify it as “programming” at all. There are no loops, function invocations, or algorithms. It is not Turing-complete by design. Implementation details are almost completely absent, leaving little to be provided other than insight about the behavior of the software.

1.5 Typing Discipline

The type system in IOP is a central concept. Languages which are said to be *Gradually Typed* (TypeScript, Hack) create an obvious distinction between two categories of code: code to be executed, and code that exists solely to inform the compiler (usually in the form of type annotations). IOP can therefore be thought of as a way of discarding the concerns of the first category completely, and only involving the concerns of the second. In this way, IOP is not so much “programming” as it is a process of making declarative statements in a kind of stand-alone type system.

This is achieved via two typing strategies that appear to be unique to IOP, *Taxonomic Typing* and *Predictive Typing*.

1.5.1 Taxonomic Typing

Most programming languages define a small number of primitives (string, boolean, integer, float), and then facilitate the composition of more complex types through classes and/or structs. Taxonomic Typing is a system that defines a large number of language primitives, and organizes them into a taxonomy according to their semantic purpose. *Description* and *Phone Number*, for example, are known data types, underneath the *Text* family. This differs from typing based on a standard subclassed common library found in most languages, as entries in the taxonomy are true language primitives that have their own data containment constraints, and supply the compiler with insight in order to reason about them intelligently.

1.5.2 Predictive Typing

Predictive Typing is a strategy for achieving type inference². Predictive Typing relies heavily on Labels.

Enabling non-programmers to provide typing information was a source of complication during our research phase. While the subjects tested had only minor trouble with a simple drop-down list containing the usual data type choices (text, number, date), as the primitives in the system became more taxonomic, comprehension degraded sharply. This is because the subjects did not comprehend the relationship (or lack thereof) between the label assigned to a particular field or table (“First Name”), and the data contained in that field as it would be classified by some abstract taxonomy (Text \rightarrow Name \rightarrow First Name). Stated another way, the subjects were not aware that machines do not automatically infer meaning.

Predictive Typing implicitly establishes the relationship between the user-supplied label, and its classification by the type taxonomy. The current implementation is purposely naive: it simply consults a static dictionary of common language patterns, and uses substring matching to draw the association. While this is surprisingly capable, a future implementation will likely involve statistical or NLP-based methods.

²https://en.wikipedia.org/wiki/Type_inference

However, regardless of how capable such techniques become, there will always be an element of guesswork, and therefore the system still allows explicit type annotation if desired.

1.6 Document Structure

Our implementation of IOP is designed to operate within a purpose-built Structured Editor³. Therefore, it lacks a grammar specification, as well as a traditional compiler front-end. We opted against a text-based language for a variety of reasons:

1. IOP apps are represented in concise documents that are typically assembled by a single person, rather than monolithic codebases spanning thousands of lines of code and involving multiple engineers. Therefore, existing tools that require code to be in a text format such as issue trackers, unit testing platforms, performance profilers, deployment automation platforms, and merge tools are unnecessary.
2. Because IOP documents are always in a “runnable” state, multiple users would be able to contribute to the same document by adding real-time multi-user collaboration functionality to the editor.
3. Structured Editors offer a degree of editing richness and abstraction of syntax that is largely unattainable via free-form text editing[1]. (It is worth noting that three members of our team were pioneers in this field with the commercially successful CSS editor known as Stylizer.)
4. Versioning needs to be tightly controlled. Typical version control systems (Git, SVN) store a sequential series of commits, and omit details of how the textual transformations between those commits actually took place. Because IOP derives the underlying data structure from the code, this intermediary detail is required in order to maximize backward compatibility and mitigate expensive schema alteration processes within the database.
5. An API will be made available so that external tools can interact with the contents of an IOP document, which would offer more simplicity than working with a parser.



 Product	
 Title	
Searchable	yes
 Description	
Maximum Length	500

Figure 2: An example of an IOP document presented in an accompanying structured editor.

IOP documents are hierarchical and reminiscent of object-oriented class definitions. Database terminology has been chosen to mitigate “class / instance” confusion which is common among non-programmers. The rules of containment are:

1. *Documents* contain *Settings* and *Tables*.
2. *Tables* have a *Label*, and contain *Fields* and *Procedures*.
3. *Fields* have a *Label*, and contain *Attributes* and then *Procedures*.
4. *Procedures* have a *Label*, and contain *Attributes*, and then *Steps*.

³https://en.wikipedia.org/wiki/Structure_editor

5. *Steps* contain *Attributes*.

6. *Attributes* contain *Formulas*.

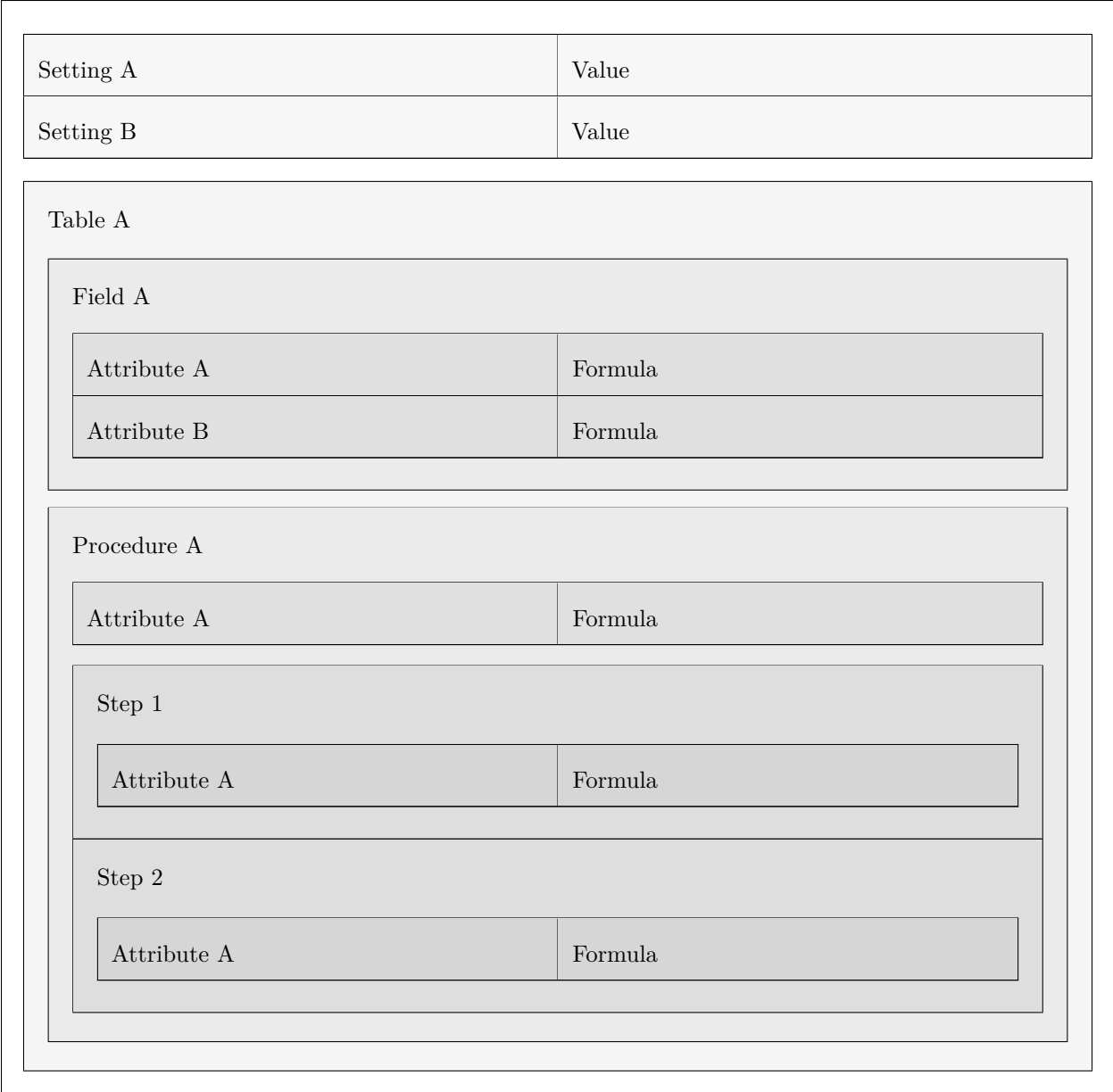


Figure 3: A visual representation of the containment hierarchy in an IOP document.

1.7 Labels

Labels are similar to identifiers, except that the actual assigned name is not arbitrary: it appears in the final output. For example, a Python application may use an identifier named `phone_number`, and then use a more friendly label “Phone Number” in the corresponding user interface. These two concepts have been merged, such that fields are now referenced by the Label (“Phone Number”) displayed in the user interface. Labels may contain multiple words ⁴, however, each word must begin with a capital letter. This allows the formula parser to differentiate between user-defined and system-defined phrases.

⁴Space characters are permitted within labels, but they may not be consecutive.

Experienced programmers may interpret this design choice as bizarre, however, one must consider that non-programmers feel that having two separate ways of referencing what appears to be same thing, to be equally as bizarre.

The problems associated with this design are mitigated through the `Display Label` attribute. As the name states, this attribute allows language elements to have a different textual label in the emitted app, than what is specified in the IOP document. This supports the following scenarios:

1. Display labels that should not be static, but rather the result of some formula
2. Display labels that do not conform to the required format (for example, “iOS” or “% Gain”)
3. Multi-language requirements

The language elements that have labels are Tables, Fields, and Procedures. The document shown in Figure 2 defines the labels *Product*, *Title*, and *Description*.

1.8 Settings

The top of every IOP document contains a preamble where settings are stored. This includes information such as API keys, the name of the app, an icon, and color scheme information.

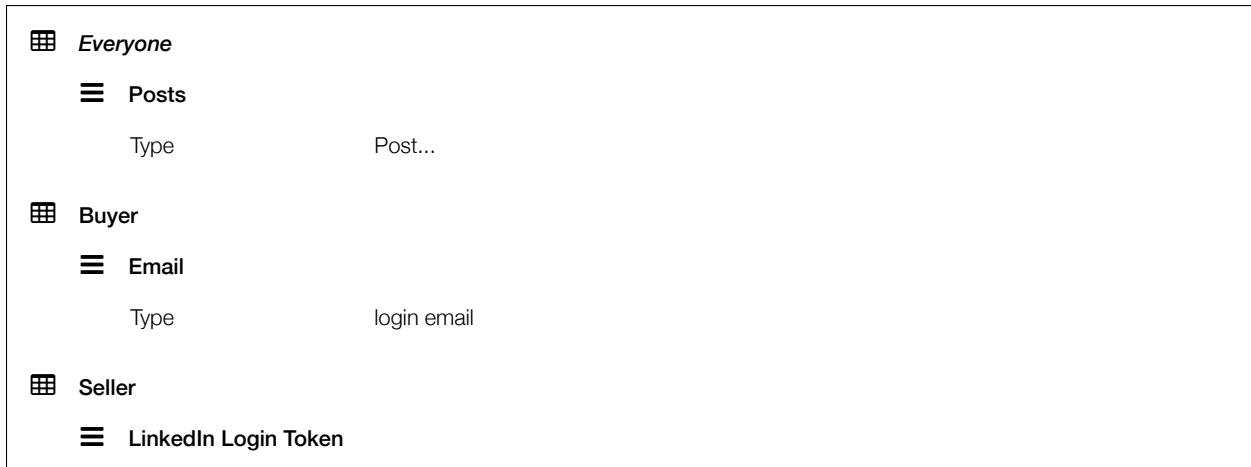
1.9 Tables

Tables are class-like grouping mechanisms that take on of three different forms:

Regular tables, which are used to represent generic domain objects within the system, for example, Products, Orders, or Reservations.

User tables, which are used to represent users. The system allows for multiple types of users. This is necessary for use cases such as marketplaces, where distinct Buyer and Seller tables would be a necessity. Tables become user tables implicitly when they have at least one field whose data type is said to cause authentication qualities to apply, such as `login email`.

The everyone table, which is an optional singleton table that can store fields on all kinds of users, even users that are not authenticated, or may not have a user account at all. The everyone table acts like an abstract base class that is extended implicitly by all user tables. A shopping cart would be a common use case for such a feature, where a user must be able to accumulate data (products), without having a user account.



Type	linkedin login token
 Post	
 Content	
Type	text

1.10 Fields

Fields may exist in two places. Most commonly, they are placed on a table to provide a means to store data of a particular type. They may also exist independently of a table as steps within a procedure. These fields hold data while the procedure is being executed, and discard this data on completion.

The data type associated with the field can usually be predicted by analyzing the label, but it may also be specified explicitly with the `Type` attribute. A type can either be a reference to an entry in the taxonomy (Phone Number, Image Price), or it may be a reference to another table defined in the document. If it is a table reference, it may either be reference a single object, such as `Product`, or a set of objects, such as `Product...`.

1.11 Procedures

Procedures are the means by which users define business rules to execute. Procedures are composed of a series of *Steps* which carry out a specific task. Although steps may specify a condition that causes them to be skipped, they are otherwise executed sequentially, never looping or recursing. The following types of steps are available:

Step	Description
Field	Captures information from the user.
Copy	Copy an object, or a set of objects from one location in the database to another.
Move	Moves an object, or a set of objects from one location in the database to another.
Delete	Deletes an object, or a set of objects.
Increase	Increments the value of the number value stored in a field.
Decrease	Decrements the value of the number value stored in a field.
Update	Assigns a new value to a field.
Pay	Initiates a monetary transaction.
Notify	Sends a push notification to the specified user.

Show	Redirects the user to a screen displaying the specified object.
Call	Invokes an API service through HTTP.
Delay	Halts execution of steps for a specified amount of time.
Quit	Prematurely terminates the execution of the procedure.

By default, procedures are executed via the user tapping a button that is placed into the app specifically to provide access to the procedure. However, procedures can be configured to execute in response to other less explicit events, through their `Trigger` attribute. This attribute can have the following values:

Trigger Kind	Valid inside...	Triggered when...
Explicit	Tables, Set Fields	the user clicks a button (which is fit into the UI automatically).
Create	Tables	an object is created.
View	Tables, Media Fields	an object or image is viewed by a user, or a video is watched in full.
Delete	Tables	an object is deleted.
Update	Tables, Fields	the data in a field is updated.
Add	Set Fields	an object is added to a set field.
Remove	Set Fields	an object is removed from a set field.

It should be noted that procedures do not execute in response to data-related events that were caused by another procedure, as this would introduce Turing-completeness into the language. For example, consider a situation where a procedure executes in response to an update of a particular piece of data, but this procedure defines a step which updates the same piece of data. Executing this procedure would result in an infinite loop of executing and updating. Future work on the system may relax this restriction for cases where it can be proven that infinite looping cannot occur.

1.12 Formulas

Formulas are *Arrow Function*-like constructs that reside within an attribute. Like the formulas of a Spreadsheet, they are guaranteed to terminate and cannot produce side effects. While formulas can draw data from other parts of the system, they cannot draw from other formulas, thereby preventing circular invocation. Formulas are composed of four language elements: *Operators*, *Pronouns*, *Literals*, and *Labels*.

1.12.1 Pronouns

IOP introduces the concept of *Pronouns*, which are shortcuts that replace well-known objects. Not every pronoun is available in the formula for every kind of attribute. The system defines the following pronouns:

user: Refers to the object representing the user that is currently logged into the system.

this: Refers to the object surrounding the formula (similar to the **this** keyword in C-family languages).

item: Refers to a single object in a set (for formulas that are intended to be applied to every item in a set).

1.12.2 Operators

Most of the features of the formula syntax exist in the form of Operators. Operators have an arity of either *nullary*, *unary*, or *binary*. They may also be either left-associative (*L-Unary*) or right-associative (*R-Unary*).

Order	Operator	Type	Symbols	LHS / RHS	Returns
1	Grouping		()	any / any	any
2	Member Access	Binary		Field	any
3	Concatenation	Binary		Option or Set	Set
	Set Transform	L-Unary	<i>(See below)</i>	Set / Set	Set
4	Exponentiation	R-Binary	^	Number	Number
6	Multiplication	Binary		Number	Number
	Division	Binary	÷	Number	Number
7	Addition	Binary	+	Number	Number
	Subtraction	Binary	-	Number	Number
8	Less Than	Binary	<	Number	Boolean
	Less Than Or Equal	Binary	<=	Number	Boolean
	Greater Than	Binary	>	Number	Boolean
	Greater Than Or Equal	Binary	>=	Number	Boolean
	Date Less Than	Binary	before	Date	Boolean
	Date Less Than Or Equal	Binary	on or before	Date	Boolean
	Date Greater Than	Binary	after	Date	Boolean

	Date Greater Than Or Equal	Binary	on or after	Date	Boolean
9	Unary Authentication State Check	R-Unary	authenticated as	Table	Boolean
	Nullary Authentication State Check	Nullary	authenticated		Boolean
	Universe-Wide List Accessor	L-Unary	each	Table	Set
	Slate-W Accessor	L-Unary	each local	Table	Set
10	Equality	Binary	is	any	Boolean
	Inequality	Binary	is not	any	Boolean
	Containment	Binary	in		Boolean
	Non-Containment	Binary	not in		Boolean
	Complete Containment	Binary	all in		Boolean
11	Inversion	R-Unary	not	Boolean	Boolean
12	Logical AND	Binary	and	Boolean	Boolean
13	Logical OR	Binary	or	Boolean	Boolean
14	Predicator	Binary	when	any / Boolean	LHS Type
	Fallback Predicator	L-Unary	otherwise	any	LHS Type
15	Statement Segmenter	Binary	(newline)	any	any

The system also defines a series of *Set Transform Operators*. These operators apply a transformation to the preceding set in the formula. Some of these operators are *Parametric*, meaning that they require an input parameter in order to execute. The complete list of operators are as follows:

Operator	Parametric Type	Returns
(where ...)	Boolean	Set
(sort ...)	Property	Set
(sum ...)	Property	Number
(average ...)	Property	Number
(size)	None	Number

(greatest ...)	Number Property	Object
(least ...)	Number Property	Object
(newest ...)	Date Property (optional)	Date
(oldest ...)	Date Property (optional)	Date
(first)	None	Object
(last)	None	Object
(reverse)	None	Set

1.12.3 Literals

The system defines literal values common in many programming languages:





Boolean: yes, no (synonyms on, off and true, false are also supported.)

String: "the quick brown fox jumped over the lazy dog"

Number: 10, 1.2, -5

1.13 Complete Example

The IOP document below is the complete “source code” for a business card exchange app. It allows users to exchange contact information using QR codes at the transfer mechanism. It is not meant to be a toy example, but rather a fully prepared app, ready for deployment and users. This specific example was chosen because the same app was built by one of the founders of Blank using traditional methods. Total project length was eight months and cost \$55,000 CAD.

	Member	
		Email
	Type	login email
		LinkedIn Login Token
	Type	linkedin login token
		Rolodex
	Pin	yes
	Type	Card...
	Visible	self
	Editable	no

Deletable	self
Searchable	self
Preview	Avatar, First Name, Last Name, Position
Sort	Last Name
Maximum Size	100000

Add A Card

Icon	plus
Location	critical

① Field

Name	Incoming Card
Type	barcode

② Copy

From	Incoming Card
To	Rolodex

Categories


Pin	yes
Type	Category...
Preview	Name
Visible	self
Editable	self
Deletable	self
Sources	My Cards
Sort	Name

My Cards

Pin	yes
Type	Card...
Preview	Company, Position
Visible	self
Editable	self
Searchable	self
Sort	Position

Maximum size 200


 **Category**

 **Owner**


Type Member

Visible no

Value user

 **Name**


Editable Owner

 **Cards**

Type Card...

Preview Avatar, First Name, Last Name, Company, Position

Sources Rolodex

 **Add A Card**

Icon plus

1 Field

Name Choose A Card To Add

Type Card

2 Copy

From Choose A Card To Add

To Cards

 **Card**

 **Add To Category**

Visible Owner is not user

Icon plus

1 Field

Name Choose A Category

Type Category

2 Copy

From this

To Choose A Category

≡ **Avatar**
Editable Owner

≡ **First Name**
Editable Owner
Searchable yes

≡ **Last Name**
Editable Owner
Searchable yes

≡ **Logo**
Type image
Editable Owner

≡ **Company**
Type text
Editable Owner
Searchable yes

≡ **Position**
Type text
Editable Owner
Searchable yes

≡ **Email**
Editable Owner

≡ **Work Phone**
Editable Owner

≡ **Cell Phone**
Editable Owner

≡ **Address**
Editable Owner
Display Format written

≡ **Barcode**

☰ **Owner**

Type	Member
Value	user
Visible	no

2 Automated User Interface Generation

IOP paves the way for mobile app user interfaces to be generated and implemented automatically. This process involves answering four basic questions:

1. What screens are needed?
2. How do these screens fit together? (i.e., what is the the navigational structure?)
3. What controls must be present on each screen?
4. What visual design cues should be used to render these controls?

2.1 Scope

It is important to note that this method is not intended to be so far-reaching as to be able to replicate the majority of mobile app user interfaces **as they exist today**. Setting aside the obvious explosion of complexity that such a requirement would cause, one would first need to consider whether this would even produce better results. Most people would agree that the design of many mobile apps is simply unintuitive and confusing. Replicating unintuitive and confusing designs is of course a non-goal. Therefore, the intention of this method is to replicate the majority of mobile user interfaces **as they could exist**, if they were to follow a standardized method.

2.2 Inferring Navigational Structure From Data Relationships

Inferring a user interface from a data model is not particularly difficult. Various commercial and open-source products have been doing this for years.

A naive algorithm is also easy to imagine. Every table in a model may be represented as a screen displaying the contents of one particular row in the table. One-to-one relationships in that row become clickable links. One-to-many relationships become scrollable lists of objects.

However, challenges arise as the requirements become less trivial, i.e. consumer-facing apps and tailored experiences. Creating software that meets such requirements while maintaining the current notion of a “data model” would be quite difficult.

This is because there is simply not enough information available in relational or object-oriented models to yield a tailored result. For example, not all controls or the screens on which they reside are a direct consequence of the mere existence of a given table or field. Consider just two common scenarios:

1. A reservation flow where the user is walked through a series of screens, some of which collect data that is not stored in the database, but rather is used as input to a third-party API.
2. A content community where the access to the moderation-related features is predicated on whether or not a given user has been assigned moderator privileges.

An interface that facilitates such scenarios could not be derived from a relational or object-oriented model alone. Therefore, just as there is a manageable “impedance mismatch” between object-oriented and relational structures, we propose that there is also a manageable impedance mismatch between the modeling of data, and the modeling of screens with their navigational relationships. This is in contrast to the industry accepted viewpoint, which is that these two processes are highly dissimilar. IOP seeks to erase this mismatch, yielding a scenario where the modeling data and the structuring of user interface are strictly synonymous, which is the key to automatic generation of specialized, high-quality user interfaces.

2.2.1 Basic Algorithm

Our strategy for inferring navigational structure begins with the concept of the *Pin Screen*. This screen is the central point from which all navigation originates. The screen is composed of a series of lists which are organized into a tab structure. The lists are derived from the fields on the table that relates to the authenticated user, where the formula specified in the `Pin` attribute evaluates to a positive value.

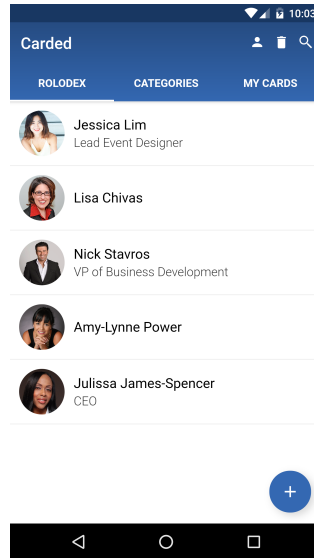


Figure 4: A pin screen on an Android device with three fields displaying as tabs.

The items in these tabbed lists are typically clickable links that advance the user to object screens that display the contents of the clicked object.

If the clicked object has fields whose type is *object* or *set*, these object screens then potentially link to deeper object screens or list screens. This occurs recursively allowing the user to navigate to (and away from) all reachable nodes in the graph.

The reachability of a given node can be augmented by formulas in attributes on a given field. For example, the `Visible` attribute on a field may generate a negative response, and thereby prevent it from being displayed to certain users, which in turn prevents access to the subtree of screens behind it. Or the screens generated to support the procedures defined on tables that collect information from the user.

2.2.2 Screen Types

The system defines a finite set of *Screen Types*, which are essentially self-configuring layouts that rearrange themselves to facilitate the requirements defined by the model.

The aforementioned List and Object screens are what the system calls *Model Screens*. Model Screens are used to display the contents of a given data entity. The Model screens are as follows:

Pinned List	Refers to a screen representing the items of a set that has been added to the pin screen.
-------------	---

Nested List	Refers to a screen representing the items of a set owned by an object.
Object (editable)	Refers to a screen that displays the contents of one particular object in the editable state.
Object (non-editable)	Refers to a screen that displays the contents of one particular object in the non-editable state.

The system defines another class of screens called *Auxiliary Screens*. Auxiliary Screens are used to support the functionality of a given model that is more or less constant across all apps, for example, authentication and search. The Auxiliary Screens are as follows:

Intro	Refers to the screens that appear to first-time users before any other screen, in order to explain the app's purpose.
Login	Refers to the screen that carries out the authentication process.
Password Reset	Refers to the helper screens found in both the login screen and the user profile, in the cases where email and password authentication is enabled.
Account Creation	Refers to screens used to create user accounts.
OAuth	Refers to the series of screens used to manage the OAuth login process.
Omni Search	Refers to a screen that handles the searching of the contents of the objects in all pinned sets.
List Search	Refers to a screen that handles the searching of the contents of the objects in one particular (non-pinned) set.
Procedure Step	Refers to the screens that are generated to support the collection of data in order to fulfill a procedure.

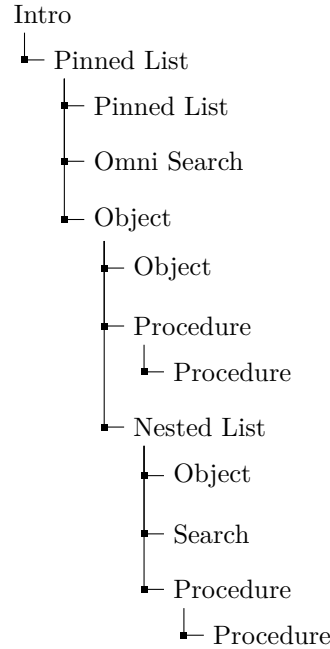


Figure 5: Representation of the main navigational structure.

2.2.3 An Abstract Representation Of Navigation

Screen types specify rules about which screens can be underneath them in their navigational substructure. These rules, when combined into a unanimous hierarchical graph, form an abstract representation of a navigation model that meets the needs of the majority of mobile apps. A somewhat simplified version of this graph is shown below.

All nodes in the main navigational structure are conditional. From an intro screen, the only reachable screen is a pinned list screen. From a pinned list screen, the user can reach another pinned list screen, the omni-search screen, or an object screen. From an object screen, the user can reach another object screen, a procedure screen, or a nested list. Procedure screens operate like a sequence, so they can only advance to other procedure screens (until the procedure is completed and the user is returned back to the starting screen).

The authentication screens conform to an independent structure, which is inserted into the flow of navigation when the user must upgrade their authentication state in order to navigate to the destination screen.

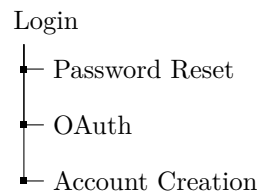














Figure 6: Representation of the authentication navigational structure.

2.2.4 Real-World Examples

IOP documents below do not describe complete apps, but illustrate possible solutions to the reservation and content community scenarios mentioned above:

	Customer
	Email
Type	login email
	First Name
...	
	Last Name
...	
	Venue
...	
	Reserve
①	...
②	Field
Label	Subscribe To Newsletter
Type	option
Default Value	yes
③	Call
Service	mailchimp
Feature	add subscriber
Email	user / Email
First Name	user / First Name
Last Name	user / Last Name

Explanation: The above document defines two tables: Customer and Venue. Customer has an Email field whose data type is `login email`, which causes Customer objects to gain authentication qualities and be treated by the system as users. The Venue table has a procedure with the label “Reserve”. For brevity sake, only two steps are shown in the example. Step 2 collects a value from the customer to determine whether to subscribe the customer to a newsletter. Step 3 calls the `mailchimp` service, specifying the `add subscriber` feature. Use of this feature then requires inputs for Email, First Name, and Last Name. The `user` pronoun stores a reference to the active user, which in this case must be a Customer object because the document only defines a single table with authentication qualities. The type of `user` can therefore be inferred as a Customer, so the Email, First Name, and Last name values can be safely drawn from `user` to populate the attributes.

	Member	
		Email
	Type	login email
		Privileges
	Type	Contributor, Moderator, Administrator
	Editable	user / Privileges contains Administrator
	Post	
		Posted By
	Type	Member
	Editable	no
	Default Value	user
		Content
	Editable	user in Posted By or user / Privileges contains Moderator

Explanation: The above document defines two tables: Member and Post. The Member table has an Email field, and a Privileges field whose data type is a definition union. These privileges are editable if the authenticated user (which is accessed through the `user` pronoun) has the Administrator definition set as a privilege. The Post table defines two fields, Posted By and Content. The Posted By field has a non-editable default value of the current user, which is assigned when the Post object is first created, which causes the field to always store a reference to the member that created the Post. The active user is able to edit the Content field if they created the Post, or if they have the Moderator definition as a privilege.

2.3 Deterministic Screen Layout Optimization

Section 3.1.2 refers to the concepts of Model Screens and Auxiliary Screens. The reconfiguration characteristics of the auxiliary screens are somewhat obvious and have therefore been omitted from this document. However, the techniques used to determine the optimal layout for the model screens is quite complex, and is covered here.

2.3.1 List Screen Layout

Lists screens are a representation of a set field. They are composed of a scrollable list of items, as well as the surrounding *Fixtures*.

The scrollable list of items is a visual representation of the items in the set, and the layout of these items is determined by the rules discussed in *Preview Layout*.

Fixtures refers to the headers, footers, and title bars that are allowed by via the guidelines of the host operating system (meaning, the Apple Human Interface Design Guidelines and the Android User Interface Guidelines). Fixtures are needed to provide UI space for links to other list screens, buttons to execute procedures defined in the model, and buttons to access system-supplied features (such as search or edit).

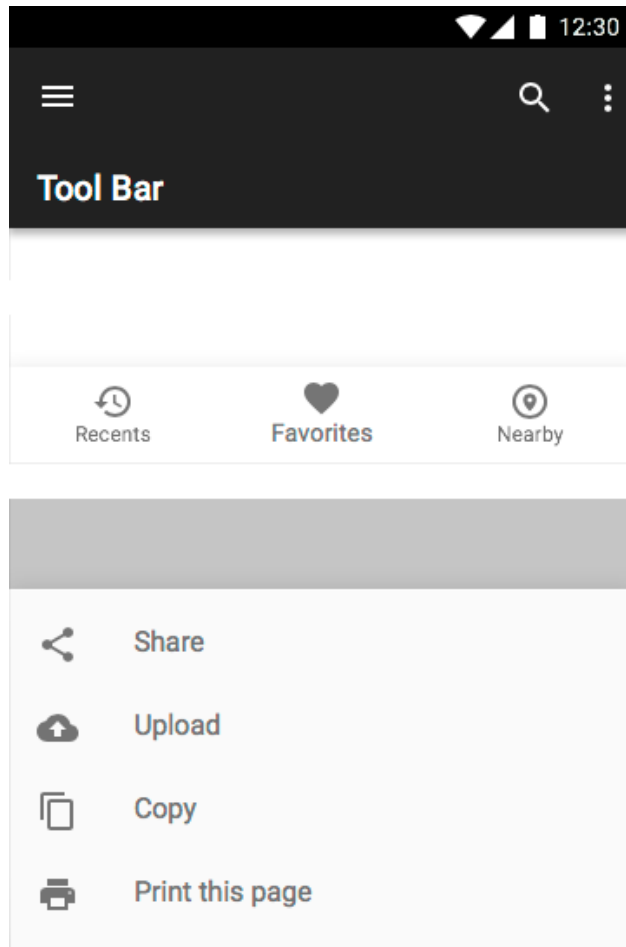


Figure 7: Examples of fixtures as supplied by the Android operating system.

The system stores a body of facts about the capabilities of each fixture, which are used to reason about which fixtures should be used on a given list screen. The basic outline of the reasoning process is as follows:

1. Determine the set of components (links, buttons, features) that must be accessible from the screen.
2. By consulting the body of facts, determine the set of candidate fixtures that may be used to provide UI space for these components.
3. Generate the set of all possible fixture combinations that fully accommodate the accessible components (screens often require more than one fixture to achieve this).
4. Compute a cost value of each fixture combination, which is a function of the quantity of screen real estate consumed, and the sum of the number of user actions required to reach each component.

The fixture combination with the lowest cost value is then used as the organizational structure in which components are placed, before being displayed to the user.

2.3.2 Object Screen Layout

Object screens can be a representation of the two main views of an object: the read-only state, and the editable state.

The method used to organize the screen in the editable state is fairly straight-forward: the components analogous to each editable field defined by the object are organized vertically (top-to-bottom) on the screen, in the order in which they appear in the IOP document. Save and cancel buttons are then positioned accordingly. Examples of analogous widgets are:

- Description field → Multi-line text box
- Photo field → Pull image from camera or file system
- Date field → Date selection or date range control
- Boolean field → Checkbox
- Object field → Search screen to locate an object of the required type to assign to the field.

Organizing the read-only state is more complex. This is because it needs to be optimized for content consumption, and must provide a way to execute the available procedures. (Note that procedures are not available while in the editable state).

Similar to the simple strategy used in the editable state, components analogous to each visible field are organized top-to-bottom. However, in this case, buttons are included to access the available procedures. The particular rendering style of the components is augmented by scanning the underlying model for heuristics. While it is beyond the scope of this document to cover the entire breadth of recognized heuristics, some examples include:

- a. Procedures whose `Dock` attribute computes to a positive value are docked to a screen edge, rather than participating in the scrolling of the screen contents.
- b. If a sequence of fields exist whose types are `Avatar`, `First Name`, `Last Name` (and in this order), the fields are presented via a centered avatar with the name shown below.
- c. If the last field defined in a table is a set, the set contents are embedded in the object screen as an indefinitely scrolling list of previews, rather than simply linking to a nested list screen.

2.3.3 Preview Layout

Previews are small regions of UI space that display fields from a related object in a carefully organized layout. When a preview is pressed, the user is transferred to an object screen that displays the full contents of the related object.

Previews are most commonly found on list screens. A list screen is the representation of a set of objects, and those objects are represented as a scrollable list of previews. Previews are also used on object screens when the object being represented has a field whose data type is another object.

Although the term “preview” is likely unique to Blank, the concept has been used widely since the dawn of graphical user interfaces.

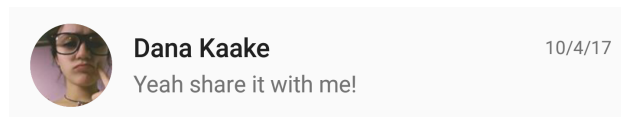


Figure 8: Example of a preview as generated by the preview rendering system.

Preview layouts are determined automatically. The model is only required to supply an ordered list of fields that should be displayed.

Internally, previews use an adaptive layout that defines a series of “slots”. Each slot defines a comprehensive set of rules that dictate what field types can occupy the slot, and under what conditions. These rules are designed to produce previews that are guaranteed to be organized in ways that are aligned with user expectations.

For example, if the preview shown in Figure 8 were to include an hourly rate field, its ideal position would most definitely not be the area where the name fields are located, but rather where the date field is located, and it would be displayed with a stronger font variant.

This means that some fields, such as file fields and barcode fields, are not permitted to exist on previews at all.

3 Data Management

3.1 Microservice Generation

In addition to compiling an IOP document into a fully functional mobile app, IOP also generates a series of microservices that support the app while it is operational.

Because IOP is a non-Turing-complete system, static program analysis can be used to determine precisely what an app will and will not do, before actually generating or executing the app. This knowledge advises the compiler on what microservices are actually needed, and their behavior.

For example, all validation logic and security requirements are essentially already specified in the IOP document, so a simple translation process can convert the formulas as specified to executable code that enforces these constraints. The data queries that can be performed are pre-generated and pre-optimized.

3.2 Distributability

Apps are currently built in such a way that the security domain of the underlying database (or set of databases) is typically aligned with the app itself. Any sharing of data to third parties almost always occurs through the creation of ad-hoc API endpoints which are guarded by an ad-hoc authentication process. Blank apps use a unique data storage method that does not resemble this monolithic architecture.

All data across the entire Blank ecosystem is stored in a common network, which allows any Blank app direct access to the data of any other Blank app, **if and only if** sufficient permissions have been assigned. These permissions are defined in IOP formulas, which allows each individual strand of data to define its own security domain, rather than having a global security domain enforced at the app level. Additionally, because IOP formulas are not Turing-complete, their security can be proven mathematically.

This highly distributed design provides a degree of data sharing and app interoperability and connectedness that would be otherwise difficult to achieve.

4 Public Interfaces

Today, web APIs are the dominant method by which independent applications communicate. The general architecture is quite simple: an enterprise provides data retrieval and data submission services that can be invoked programmatically by outside applications. This model has served the software ecosystem well, allowing many apps to communicate with one another to share data and resources. Blank apps therefore have the ability to communicate with third party web APIs (via a `Call` step within a procedure). However, web APIs have limitations:

1. An organization that wishes to provide a web API must be willing to commit to a considerable amount of upfront and ongoing design and engineering work.
2. Web APIs only provide a means for communication between systems. However, in many cases, a supporting user interface required to actually facilitate this communication. This user interface is typically built by the consumer of the API, although some vendors provide software libraries to ease this burden.

Public Interfaces are an alternate means to achieve sharing and collaboration between apps that goes beyond mere data exchange between disparate backend systems.

App creators can move parts of their IOP document into a public interface, which allows entire sections of their app (data, features, and supporting interface), to be replicated within other apps. This allows creators of other apps to take a part of a screen, an entire screen, or a series of screens and embed it into their own apps. It is worth reiterating that this is made possible because IOP erases the separation between the structure of data, and the corresponding UI used to access that data, such that these are essentially *are one and the same*.

4.1 Formats

Public interfaces are either *Broadcasts* or *Crawlers*.

A **Broadcast** allows an app creator to make aspects of their app available for other apps to embed. For example:

- a. An app resembling Twitter could make its tweet data (and a supporting interface) available for third parties to embed into their own apps.
- b. An app for an independent hotel could embed the ordering system for a nearby pizzeria, allowing guests to order pizza directly from their app, and have it delivered directly to their room.
- c. An influencer could monetize their app (and their content) by embedding the product catalog of a trusted store that sells related products.

A **Crawler** is the reverse of a Broadcast Interface: it allows an app creator to consume the data of other apps. Crawler Interfaces are built by app creators, who put forth a prescription for a specific structure consisting of tables, fields, procedures, and attributes in the form of a contract. Outside apps may choose to participate in this contract by including the language elements prescribed (and potentially passing an verification process).

Crawler Interfaces are therefore similar to web crawlers that consume other web pages. However, whereas web crawlers consume unstructured content of other webpages, and usually doing so without explicit permission, crawler interfaces consume content with a very specific structure, and typically involve mutual agreement by both parties. This creates opportunities for styles of apps which would be difficult to create or market in the current technological landscape.

For example, an Instagram-like app could create a crawler interface that prescribes a foundational structure: posts containing images of a particular size, the capacity for comments, and the ability to “like” a particular post. Then, thousands of other influencers (each possessing with their own app), could implement the interface as prescribed, and have the conforming portion of their app discovered by the Instagram-like app that created the crawler interface.

However, the influencers from the example could potentially participate in many crawler interfaces, which creates a distributed app network where content discoverability is not controlled by any single authority.

4.2 Polymorphism

Because IOP is a non-Turing-complete system, it is possible to determine the exact range of values that a given formula may generate (introducing Turing-completeness would make this undecidable). This allows a static analysis process to determine the polymorphic compatibility (or lack thereof) between two public interfaces.

For example, consider a public interface A that calls for a `Title` field with a `Maximum Length` attribute whose formula returns 100 (meaning 100 characters). If a public interface B defines a similar field, except with a formula returning 50, public interface B can be deterministically deemed a mathematical subset of public interface A.

This aids in the versioning of public interfaces, on part of the owner. Changes can be made to public interfaces even while they are online with active participants, however, all successive versions are required to be a structural superset of the previous version, to ensure backward compatibility.

This also aids in the adoption of public interfaces, on part of the implementor. Public interfaces do not need to be implemented exactly as prescribed. Instead, the range of possible values stored in the prescribed fields must only be a subset of the prescribed ranges.

5 FAQ

What is the process for testing apps during development?

Blank app binaries that do not vary significantly from one app to the next. In fact, the only differentiating factor is the actual IOP document (which consumes not more than 1-2KB), and any non-standard assets. IOP documents therefore act like a blueprint that controls the behavior of a thick runtime. The IOP document and the underlying runtime are completely decoupled, even as the app is running. It is therefore possible to react to a user action by replacing the IOP document loaded into the system with another IOP document, and have the app instantaneously reconfigure itself into a completely separate app.

Blank apps can therefore be developed in real-time. A special Blank app will be made available that has no document loaded, but rather streams documents directly from the structured editor, and reconfigures itself on-the-fly.

How do Blank apps incorporate specialized algorithms?

Procedures may include a `Call` steps, which cause HTTP calls to be made to arbitrary endpoints. Specialized algorithms can be written, deployed to proprietary backend systems, and invoked through procedure steps.

What if an alternate UX is desired that does not directly coincide with the data model?

End-user confusion is almost always results when either the data model or the navigational flow of an app does not coincide with the problem domain. As a general rule, all three of these concerns should be kept in alignment.

There are of course many counter-examples where this rule does not apply, so Blank provides enough attributes to control the rendering of various language constructs (such as `Visible` to hide sensitive data and `Display Format` to control the rendering style of an address or a unit value). Cases where deviation is warranted *and not possible* should therefore be rare.

Can a seemingly small change to an IOP document cause a drastic reworking of the UI/UX?

No. While some changes will cause a more pronounced effect than others, it should be noted that:

- a. the navigational structure of an app is primarily a function of the structure of the underlying IOP document.
- b. the layout of each screen is primarily a function of the order in which fields are defined in their containing table.

Therefore, changes to an IOP document of a given magnitude will result in changes to the resulting UI/UX of a similar magnitude.

When a public interface changes, are all implementing apps updated automatically?

The particulars are still under consideration. Most likely, implementors will have control over over how (or if) successive versions of a public interface are implemented automatically in their apps.

6 Conclusion

We present a new method of creating software called Insight-Oriented Programming. Insight-Oriented Programming operates at the level of abstraction that non-programmers think, by shifting the focus of software development towards providing insight into end-goals. This insight is then used to generate a mobile app, automatically. This style of “programming” results in a dramatic reduction in the amount of human

effort required to produce software, and as a side effect, a dramatic reduction in the amount of data required to store software produced in this way, and to transmit it to devices.

This reduction in app size, coupled with public interfaces, allow the boundaries between apps to become blurred. This paves the way for a new kind of internet. An internet where all nodes are native mobile apps created with minimal effort, and all provide beautifully consistent user experiences.

References

- [1] Altadmri, Amjad & Kölling, Michael & Brown, Neil. (2016). The Cost of Syntax and How to Avoid It: Text versus Frame-Based Editing. 748-753. 10.1109/COMPSAC.2016.204.