Escaping the Maze of Twisty Classes

Sean McDirmid
Microsoft Research Asia
Beijing China
smcdirm@microsoft.com

Abstract

Programmers demand more extensive class libraries so they can reuse more code and write less of their own. However, these libraries are often so large that programmers get lost in **deep** hierarchies of classes and their members that are very **broad** in number. Yet language designers continue to focus on computation, leaving tools to solve library exploration problems without much help from the language.

This paper applies language design to improve IDE code completion that enables in-situ library exploration. **Inference** tackles depth by listing completions as long as the program can be "fixed" to support their selection; e.g. "pressed" can be listed as a widget completion since a widget can be a button. **Influence** mitigates breadth by leveraging types as completion selection models; e.g. a pressed event is more likely to be used on a button than a mouse event. We apply this design to YinYang, a language for programming simulations on tablets using touch-based menus.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Code Completion, API Recommenders

1. Introduction

Programmers expect a programming language to be supported by a rich and extensive class library; i.e. it must come with its "batteries included" [27]. But as libraries grow larger, they become impenetrable "mazes" of **deep** name hierarchies and structural dependencies with **broad** sets of abstractions to choose from. To cope with these mazes, programmers increasingly augment library documentation with *code completion* to explore the library by leveraging context, such as type information, of the edited code. Unfortunately, code completion is limited by library depth that obscures what choices are visible in a code completion menu; e.g. a method call is not listed as a choice in a code completion menu for an object when the method's defining class is not yet extended by the object. Besides depth, programmers can also be overwhelmed by very broad code completion menus that are not efficiently scanned; e.g. a `Button` object's code completion menu in Microsoft's Windows Presentation Foundation (WPF) lists over 290 choices [23].

Contrast code completion to modern web search that ignores depth and ranks broad results based on relevance; e.g. via Google's PageRank [28]. Web search has led to the viability of *long tail* niche content that caters to diverse user interests [1]. Programs also have diverse needs, and so we could also benefit from a long tail of library functionality if it could be easily searched. Many tools attempt to bring the search experience to programming; consider code search engines [29, 30, 32], code and API recommendation systems [9, 15, 18, 37], or better forms of code completion [5, 13, 15, 31]. However, such tools are limited in their effectiveness by current programming languages where hierarchy is difficult to bypass and relevance is ill-defined.

Beyond building better tools, this paper argues that we should also design our languages to enable better library exploration. As a case in point, types can enhance the code completion menus that help us write code rather than just ensuring that we write type safe code; i.e.

> *Ask not what your code can do for your types, but what your types can do for your code!*

Types already provide semantic feedback that enable code completion in the first place; we expand on this support in two ways. First, types can prescribe the **inference** of program structure so that abstractions can be used safely out of order, reducing the impact of depth on code completion menus. So rather than just use types to restrict that "code can only call the is-checked method on checkbox objects," inference allows types to also prescribe that "objects become checkboxes when code calls the is-checked method on them." Because a program does not need to be "prepared" to use something, code completion menus can list all choices that do not lead to inconsistent constructions. Second, types can express **influence** relationships so that code completion

Onward! 2012, October 19–26, 2012, Tucson, Arizona, USA.

menus can rank broad choices. For example, although "the resize method can be called on UI label objects," label sizes are typically fixed to content; and types should also express that "UI label objects are rarely resized" to deemphasize the `Resize` method in a `Label` object's code completion menu.

Inference eliminates depth at the expense of creating more breadth that influence then manages through ranking. Choice rankings in code completion menus progressively adapt as object types become more specific due to selections and inference. Both inference and influence are more useful when they can operate on a graph of objects, rather than one object, to express things like "setting the position of a widget causes its containing panel to become a canvas" or "widgets contained in a canvas are often resized manually."

We designed inference and influence into our experimental YinYang [19] programming language where code completion menus are essential to programmer productivity as YinYang supports programming on tablets where typing names is not very efficient. YinYang targets simulations, such as games, with a library that enables objects to easily take on niche functionality on demand; e.g. game scoring or life points. More generally, our work demonstrates that tackling the problem of finding library functionality in language design is both possible and useful.

The rest of this paper first describes inference and influence in Sections 2 and 3 to address the problems of depth and breadth. Section 4 presents our experience in using these ideas in YinYang, while interesting semantic and implementation details are discussed in Section 5. Related work is surveyed in Section 6 while Section 7 concludes.

2. Depth and Inference

Almost all modern programming languages support some form of hierarchical naming and construct (member) organization, which we refer to as *depth*. Members are only visible under strict preconditions; e.g. if the program includes a module; if a namespace is imported; if an object extends a class; and so on. Code completion can often only list members that can be accessed directly and safely from the qualifying context of an edit; e.g. the members of a WPF [23] button object are listed in a C# code completion menu [21] only if the type of the qualifying reference extends the `Button` class. The use of code completion as an exploration aid is then severely limited because it can only complete one identifier at a time, and decisions made at other points in the program, such as what class is extended by an object, determine what members are listed as choices.

Code completion first appeared as an aid in syntax-directed editors [33], e.g. Alice Pascal [34], where the emphasis was on "always correct" program input. Syntax-directed editing itself failed primarily because editing was restricted to "always correct" constructions [24], a problem fixed in today's free-form language-aware editors. However, code completion as implemented in language-aware editors

```
TRAIT Widget;
TRAIT Button  :: Widget { EVENT Pressed; }
TRAIT Slider  :: Widget { VAR   Value : double; }
TRAIT Bordered : Widget { VAR   Thickness : Px; }
```

Figure 1. Definitions of widget traits used to construct user-interface elements; A single colon (`:`) expresses mixin extension while the double colon (`::`) expresses exclusive extension; `VAR` defines a mutable property.

(starting with [20]) is still only effective when program symbols are defined in a rigid top-down order, leading to the depth problem of this section. Error-tolerant heuristic compilers in IDEs can interpret incorrect code to provide feedback and propose fixes, but their effectiveness is limited by incorrect code that is not semantically well defined.

We propose loosening up the type system of the language itself with types that can prescribe the *inference* of program structure when abstractions are used out of order. Compared to type inference [7] that infers type annotations, our inference infers code that drive behavior. An abstraction can be used, and so listed by code completion, before its type context is satisfied because inference can add code to complete it. This section discusses inference through a hypothetical Scala-like [26] language; Section 4 later describes YinYang as a concrete language that is designed around inference.

Trait Inference

Inference turns the typical restriction that "what an object can do depends on the classes it extends" around to "what classes an object extends depends on what it does;" i.e. an object will extend a class implicitly if it can access the class's functionality. For example, if an object accesses `Button`'s `Pressed` member, then the object implicitly extends the `Button` class via inference. A code completion menu can then list `Button` members, including `Pressed`, as type safe completions on any object that could extend `Button`.

Rather than focus on classes, we consider *traits* [6] that support a flexible form of mixin-like [3] linearized multiple inheritance. The fine-grained modularization of functionality into traits becomes much more usable with inference because traits do not always need to be extended explicitly and client programmers can often be oblivious to their existence. Niche functionality can be packaged up into traits that are then inferred when the functionality is selected, which we refer to as *trait inference*. Consider the traits defined in Figure 1 and the following code that uses these traits:

```
VAL w = NEW Widget;
w.Pressed -> { w.Thickness = 10 PX; }
```

The first line here creates a generic `Widget` object called "`w`." The second line accesses the `Pressed` event, inferring that the `w` object extends the `Button` trait, and assigns to its `Thickness` property (`VAR`), inferring that the `w` object extends the `Bordered` trait. The `->` operator on this line indicates con-

ditional execution over the result of a method or event; e.g. that the `Pressed` event has occurred. The programmer can be oblivious to the existence of the `Bordered` and `Button` traits; instead they only need to find the `Pressed` and `Thickness` members desired for the object, which are safely listed in `w`'s code completion menu.

Given trait inference, name binding cannot be relied on to resolve object members as traits are not first extended to establish the appropriate name scope context, and anyways, as Edwards observes: "names are too rich in meaning to waste on talking to compilers [10]." Instead, abstractions can be provided directly via the editor that deals with abstraction presentation and input as a separate concern. Beyond being crucial to enabling inference, eliminating name binding from the language also solves the name depth problem as namespaces are not needed. We discuss naming more in Section 4; for now assume traits and members are named globally.

The example in Figure 1 uses two kinds of extension operators: colon (`:`) as a typical mixin extension operator; and double colon (`::`) as an *exclusive extension* operator that suppresses multiple inheritance to disallow nonsensical trait combinations. Exclusive extension forces single inheritance on the specified extended trait; e.g. because the `Button` and `Slider` traits exclusively extend the `Widget` trait in Figure 1, an object cannot extend both of them. Exclusive extension is essential for tempering the implicit power of trait inference: without it, code completion menus could list functionality that is inconsistent with previous programmer selections. Consider the following incorrect code:

```
w.Pressed -> { ... }
w.Value = 42;
```

The `w` object cannot access both `Pressed` and `Value` members since it would have to be both a `Button` and a `Slider` to do so; as soon as the programmer selects `Pressed`, the `Value` choice is no longer visible in a code completion menu.

Slots

Programs consist of multiple objects whose collective structure can determine the capabilities of each; e.g. a WPF [23] `FrameworkElement` object supports manual positioning with `Canvas` attribute properties. Unfortunately, code completion in C# is not useful in discovering this functionality because the enabling methods are static in the `Canvas` class and not instance methods of the framework element object being positioned. Additionally, no type relationship in C# exists between a framework element object and the panel object that contains it—setting the position of an object that is not contained in a `Canvas` object is silently ignored!

Objects generally have well-defined relationships with other objects; e.g. a panel "contains" a widget, or a line connects two points. Such relationships are often encoded through field assignment or binding and then ignored by the type system. We observe that being bound to a field could be a type in itself; i.e. for `a.b = c`, `c` can be enhanced with a type of the form "that which is bound to `a.b`." Such a

```
TRAIT Panel;
TRAIT Canvas :: Panel { }
TRAIT Dock :: Panel { }
TRAIT Widget { SLOT container : Panel; }
TRAIT WidgetInCanvas : Widget {
  REFINE container : Canvas;
  VAR Position : Point;
}
TRAIT WidgetInDock : Widget {
  REFINE container : Dock;
  VAR Docking : Cardinal;
}
EVENT Tapped : Canvas {
  OUT AtTapped : Point;
}
```

Figure 2. Definitions of panel and widget traits that allow for positioned widget nesting.

binding is then not only something to be type checked, but also contributes type information to the program. We refer to fields whose bindings contribute type information as *slots*.

Unlike fields, slots can undergo narrowing type refinements as their defining traits are extended, which is safe because trait inference works across slot bindings: if a slot is refined in one trait but bound according to its old type in an extended trait, the refinement in the extending trait propagates across this slot binding to the bound object. Member selection on one object can then cause other objects in the program to implicitly extend traits via trait inference.

As an example, consider the traits defined in Figure 2 where `Panel` objects contain `Widget` objects. The `Panel` trait is exclusively extended by both the `Canvas` trait, which supports manual widget positioning, and the `Dock` trait, which supports cardinal north, south, east, and west positioning. The `Widget` trait defines a `container` slot that is meant to be bound to the panel that contains a widget object. In both the `WidgetInCanvas` and `WidgetInDock` traits, the `container` slot is refined to extend `Canvas` and `Panel` respectively while `Position` and `Docking` properties are defined to support the position scheme of each panel. Accessing either latter property can then cause the widget's container to implicitly extend the corresponding panel trait. Consider:

```
VAL p = NEW Panel;
VAL w = NEW Widget;
w.container = p;
...
w.Position = (10px, 300px);
```

This code creates `Panel` object `p` and a `Widget` object `w` where `w`'s `container` slot is bound to `p`. When the `Position` property is accessed from `w` in this code, two inferences occur: first, trait inferences causes the `w` object to implicitly extend the `WidgetInCanvas` trait that contains the `Position` member; and second, trait inference propagated through slot binding causes the `p` object to implicitly extend the `Canvas` trait

because of the WidgetInCanvas trait's extension constraint on the container slot. Beforehand, the Position property is visible in the code completion menu because w's container binding could possibly extend Canvas. Exclusive extension prevents both the Position and Docking properties from being accessed on the w object, as doing so would cause the p object to incorrectly extend both Canvas and Dock traits that both exclusively extend Panel. The two WidgetIn* traits define functionality related to specific object graph topologies, but programmers can be completely oblivious to the existence of these traits and focus just on their functionality.

Behavior Inference

To get even more mileage out of inference, we treat method usage similar to trait extension with respect to the type system. For example the Pressed button event in Figure 1 could be expressed as:

```
TRAIT Button :: Widget;
EVENT Pressed : Button;
```

meaning that button objects actually access the Pressed event by "dynamically" extending it. The same syntax is used to define a similar Tapped event in Figure 2, which extends the Canvas trait. When the canvas is actually tapped, extension of the Tapped event "fires" and the AtTapped output is available for use; consider the following code:

```
VAL p = NEW Panel;
p:Tapped -> {
  VAL w = NEW Widget;
  w.container = p;
  w.Position = p.AtTapped;
}
```

This code creates a Widget object (w) wherever the user clicks on a Panel object p. Given their treatment as extensions, methods are then subject to *behavior inference* that is very similar to trait inference. Consider:

```
VAL p = NEW Panel;
NEW Widget {
  container = p,
  Position = p.AtTapped
};
```

This code is equivalent to the previous code snippet, where AtTapped can be discovered in a code completion menu for binding Position without the programmer first selecting the Tapped event. The block that follows the NEW Widget expression initializes an anonymous object by binding its container and setting its position. Because the object's position is assigned to the AtTapped output slot of the p object, extension of the Tapped is required before the object is created. Additionally, a new and distinct object is created for each firing of a Tapped event and the w object must only be used in the context of this event firing, and therefore the object cannot be named in this snippet. Syntax used in Section 4 makes scoping restriction more obvious in the presence of inference.

Inference appears problematic in many areas: its implicit nature detracts from programmer awareness and control, while the lack of scoped name binding makes it unclear how code is inputted and how names are disambiguated in what is essentially a global namespace. Such user interface issues are addressed in Section 4. Additionally by eliminating depth, the problem of breadth is exacerbated where code completion menus are much more likely to be too broad, which we deal with next.

3. Breadth and Influence

Code completion menus with too many choices detract from library exploration as breadth makes scanning menus for desired functionality more difficult. Consider that over 290 choices are accessible through an object that extends WPF's Button class [23], where code completion [21] displays nine elements at a time in the code editor, leading to more than 32 menu pages to scroll through. An exploring programmer is also probably unable to name or filter what they are looking for, while the problem becomes worse when depth is eliminated in Section 2.

Breadth can be dealt with by adding depth to the completion menus through sub-menus that categorize choices semantically. Unlike the depth described in Section 2 that hide choices completely, sub-menus still allow a choice to be found via menu navigation. Such semantic categorization can be encoded explicitly; e.g. various visual object properties and methods can appear in a Visual category independently of their defining traits. However, we must still make a tricky trade-off between menu breadth and depth: broad menus suffer from long scan times that are at best logarithmic (if ordered by a desired key) and at worst linear (see Hick's law [12]). On the other hand, although deeper menus are faster to scan at each level, efficiency decreases rapidly as users lose focus by navigating to a new menu level [22, 25]. As a result, reducing menu depth will improve the programmer's experience. This can be accomplished by promoting choices from sub-menus to parent menus based on the likelihood that the programmer will select them, increasing menu breadth in a very targeted way.

Choice selection in a code completion menu can be estimated by a *selection model* that uses the programmer's context, primarily in the form of the code they are writing, to rank choices based on what they are likely to select next. For example, consider a Resize method of a Widget trait. Many kinds of widgets, such as buttons and labels, have preferred sizes computed based on their content and so are not often resized manually. On the other hand, Composite widgets, which enable panel nesting, do not have a preferred size and so programmers will almost always resize them. A model should then boost the Resize method's rank for objects that extend the Composite trait and reduce it for objects that extend the Button or Label trait.

A selection model can either be crafted by hand or by analyzing existing code using some kind of rank inference algorithm. Although generating a model automatically requires no manual work, existing code often does not exist in quantities needed for this approach to work effectively, especially if the language or library is new. Crafting a model by hand is often more appropriate since it does not rely on a large body of existing code to work effectively, but can also burden library designer with lots of extra work.

Influence

Given that types are already formal models of correct program structure, its not a big leap to have them act as selection models as well; i.e., types already express necessary is-a relationships, but they could also express *unnecessary* is-probably relationships as well, making it straightforward for library designers to encode selection models. For example, we could encode not only that `Canvas` must be a `Panel`, but also that a `Panel` is probably a `Canvas`. We refer to this encoding as *influence*, which complements inference by using types to express selection models.

Influence enhances the syntax of trait extension to include two extension operators divided by a slash of the form `TraitB opAB/opBA TraitA`: the first left operator (`opAB`) expresses an inverse extension relationship from `TraitA` to `TraitB` that is often unnecessary, while the second extension operator (`opBA`) encodes the typical extension relationship from `TraitB` to `TraitA` that is often necessary. Consider the `Canvas` trait definition from Figure 3:

```
TRAIT Canvas +/:: Panel;
```

This extension expresses that (a) `Panel` is often (`+`) a `Canvas`, and (b) `Canvas` is (`::`) a `Panel`, exclusively. Without the slash syntax, these two extension relationships could be expressed using standard extension syntax as:

```
TRAIT Canvas :: Panel;
TRAIT Panel + Canvas;
```

However, we use slash syntax since it allows extension definitions to be acyclic even if the actual extensions are not. Even though the extension graph is now cyclic, necessary extension edges (formed by `:` and `::`) must always form acyclic sub-graphs.

Trait definitions shown in Figure 3 encode influence using the extension operators that are listed in Figure 4. Each extension operator (α) has a unit influence ($\phi(\alpha)$) that is used by traversal of the extension graph to compute a summed "influence" that approximates how likely a trait (or method) will be extended by an object. The necessary operators (`:` and `::`) have the strongest unit influence possible (0); lower-tier operators (`-`, `--`, `---`) then express uninfluential choices; higher-tier operators (`+`, `++`, `+++`) express more influential, but still not necessary, choices; and a `*` operator indicates a neutral typical influence. The influence of a trait that can be extended by an object is computed as the sum of all edge influences in the shortest path from the object to this trait in a cyclic directed extension graph. Traits that are already necessarily extended by an object always have an influence of zero given that each `:` edge has an influence of zero. Unnecessary operators have unit influences that increase exponentially to prevent stronger (shorter) influences from quickly reducing into weaker (longer) influences as multiple edges are traversed in a shortest path. For example, a path with less than four `+` edges results in a influence that is stronger than a path with one default `*` edge.

The extension graph formed by the definitions in Figure 3 is shown in Figure 5. As mentioned before, we condense both edges of an extension into one for readability, and so the extension graph is then actually cyclic, but only the minimum influence of a trait is relevant and so efficient traversal is possible according to a shortest path algorithm. As an example, given an object `w` that only explicitly extends `Widget`, the influences for each of the traits that it can extend in Figure 3 are as follows:

```
TRAIT Panel */:: ROOT;
TRAIT Widget */:: ROOT {
  SLOT Container */: Panel;
}
TRAIT Button -/:: Widget {
  REFINE Container +/+ Canvas;
}
TRAIT Composite -/:: Widget {
  REFINE Container +/* Dock Panel;
}
TRAIT Scroll Bar --/:: Widget;
ACTION Resize -/: Widget +/- Composite {
  IN To */: Vector;
}
TRAIT Canvas +/:: Panel;
TRAIT Dock Panel -/:: Panel;
```

Figure 3. Trait and action definitions that express influence relationships; the `*`, `-`, `--`, and `+` extension operators indicate unnecessary extensions of varying influences.

operator α	description	unit influence ($\phi(\alpha)$)
`::`	exclusive	0
`:`	extended	0
`+++`	most influential	$4^0 = 1$
`++`	very influential	$4^1 = 4$
`+`	influential	4^2
`*`	default	4^3
`-`	uninfluential	4^4
`--`	very uninfluential	4^5
`---`	most uninfluential	$4^6 = 4096$
NA	impossible	∞

Figure 4. Extension operators and their unit influences.

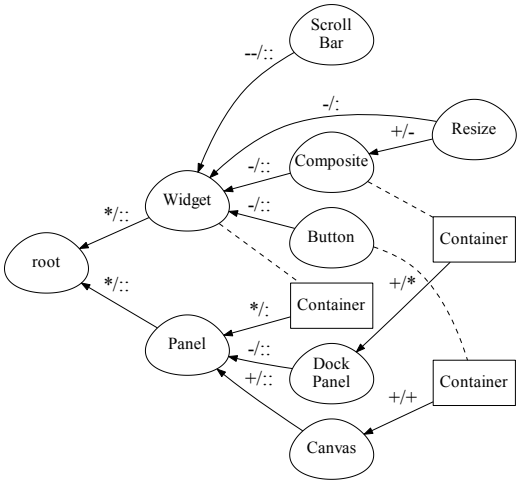


Figure 5. The extension graph for the definitions in Figure 3; traits and methods are rendered as eggs; encoded extensions are directed edges that are labeled by two extension operators; and slots are rectangles that are connected to their defining or refining traits by dashed undirected edges.

`Widget`	0
`Button`	$4^2 + 4^2 = 32$
`Composite`	4^4
`Scroll Bar`	4^5
`Resize`	$min(4^4, 4^4 + 4^2) = 4^4$

As described shortly, the `Button` trait has a summed influence that is reduced because of slot extension. There are two non-cycling traversal paths from the `Widget` trait to the `Resize` action in Figure 5: one through a direct edge and one via `Composite`. The direct path is then chosen for the influence of `Resize` given that it has a stronger influence. If object `w` later explicitly extends `Composite`, the influence of the `Resize` action is adjusted to match the stronger (shorter) influence path through `Composite` (4^2).

Slots and Influence

Extension refinements to a slot (Section 2) inherited by the trait are considered when computing its influence. Consider that `Button` objects are often placed in `Canvas` objects but not in `Dock Panel` objects given that the latter panel is mainly meant to nest larger widgets such as composites. As a result, `Button` is an uninfluential (`-`) extension for a `Widget` object unless the object's container slot is bound to a `Canvas` object. Since the `Canvas` trait is an influential (`+`) extension for a base `Panel` object, the influence for the `Button` trait in the previous table is $4^2 + 4^2$ rather than a less influential (`-`) 4^4 as its direct edge to the `Widget` trait would suggest. If `Canvas` becomes necessarily extended by the widget's container object, then `Button` becomes an influential extension (4^2) for the widget object. However, if instead the containing panel object necessarily extends the `Dock Panel` trait, the `Button` trait then becomes an uninfluential (4^4) extension as the shorter influence path through `Canvas` in the extension graph is eliminated via exclusive extension, while `Composite` becomes an influential (4^2) extension of the widget object.

Influences computed from the extension graph approximate influence of a trait or method with respect to current structure of the object graph. Incorporating the object graph into the extension graph is non-trivial: additional edges in the extension graph must be synthesized via an iterative algorithm according to the topology of the object graph; we discuss this synthesis in Section 5. Overall, these influences are then used to rank choices in code completion menus: more influential traits (or methods) are shown unobscured in code completion menus before less influential traits, which are more likely to be buried in sub-menus (Section 4).

The inclusion of influence results in a richer language for expressing extensions that is not entirely different from the one we already use in conventional object-oriented languages. On the other hand, choosing appropriate extension operators requires a lot of reasoning by the library designer on how the library will be used. Data mining could help: rather than analyze existing code to construct models automatically, usage patterns identified in the code bases, even for different libraries for different languages, can inform what extension relationships are useful. The examples in this section are informed by common patterns in existing WPF [23] programs, though our analysis so far has been very informal. We provide a more complete experience on how to design with inference and influence in the next section.

4. YinYang

We now explore how the ideas presented in Sections 2 and 3 are realized in the design of YinYang, which is a language for programming with touch on tablets [19]. Because touch-based text input is inefficient, YinYang relies heavily on code completion menus for decent programmer productivity. As mentioned in Section 2, traits and methods share the same extension semantics and are subject to inference and influence; collectively we refer to them as *tiles* that are only distinguished by whether their extensions form object attributes (traits) or cause objects to do something (methods). YinYang's programming model is described in [19]; we focus our discussion here on code input and review this programming model only where necessary. YinYang focuses on building simulations such as animated art or games, and so its code is organized into *behaviors* [4, 16], rather than blocks, that execute both reactively and autonomously. YinYang's library of tiles that support simulations is encoded in C# according to a form that resembles how traits were defined in Sections 2 and 3; this library is then presented by a graphical programming environment where programmers express simulations. The rest of this section focuses on the

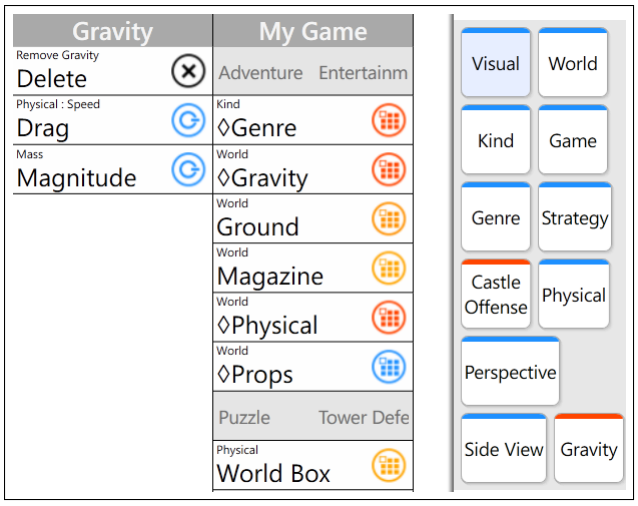
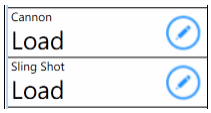


Figure 6. The YinYang editor after `Gravity` has been selected for the `My Game` object; code input menus are on the left; `My Game` labels the object's root menu; `Gravity` labels the `Gravity` sub-menu; and traits extended by the `My Game` object are on the right. Note that the screen shots in this section are scaled down to half of their actual size.

programmer's experience in using this library through the environment.

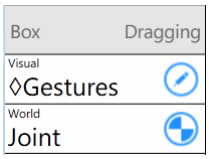
Input Menus

Figure 6 shows YinYang's programming environment in use. Code completion menus, which we refer to as *input menus*, on the left are used to select traits for the current object to extend (top right) or to cause behavior in its code section (not shown), according to what element is currently selected on the right. As discussed in Section 2, tiles are selected directly by the programmer and therefore names are a concern of the editor rather than the language. So that the programmer can disambiguate between tiles in input menus, a tile has both a *primary name* and a *secondary name*, which is often the name of another related tile. The primary and secondary names together uniquely identify the tile to the programmer in a global context so that they can distinguish tiles with the same primary names presented in the same menu. Within a menu entry, the primary name is presented prominently while the secondary name is presented above it in the periphery; consider the following tile selection menu entries:



Both tiles are called `Load`, but the programmer can use the periphery secondary names to decide if they want to load a sling shot or load a cannon. To the right of a tile's name is an icon that describes its purpose, e.g., trait, action, event, slot, and so on, which expresses computed influence with color coding from hot (more influence) to cool (less influence).

A menu presents all selectable tiles in alphabetical order for efficient by-name access. However, only a fixed number of tiles are initially visible in the menu while other entries are hidden under expandable alphabetical *elisions*. Abstract (non-selectable) implementation-free tiles express the semantic categories that were described in Section 3; tiles simply extend these *category* tiles to be organized under them. Categories are prefixed with a ◊ and organized in the menu with tile selections. Consider three input menu entries:



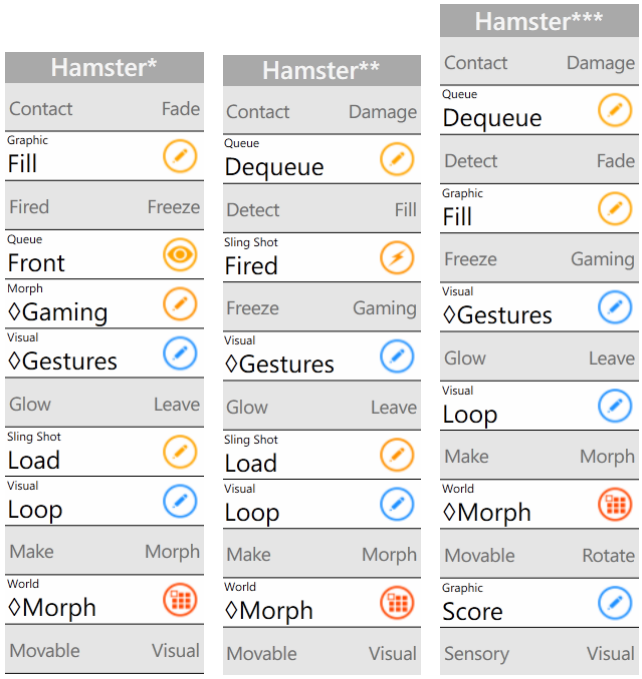
The first entry is an elision whose selection expands reveals entries with names between "Box" and "Dragging," the second entry is a semantic `Gestures` category whose selection opens a sub-menu with just gesture-related tiles, and the third entry is a `Joint` tile that can be selected for extension. Both elisions and categories are recursively defined: an elision expands to only reveal at most five unelided entries and may contain additional elisions; while a category sub-menu opens to reveal elisions and categories mixed in with unelided tile selection entries.

Inference, Influence, and Slots

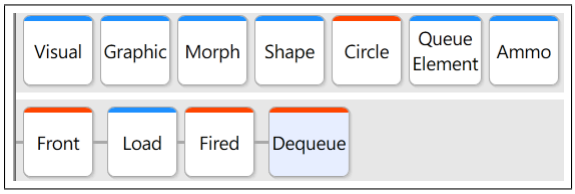
Each menu level from a root ("all") category is populated with the categories that directly belong to the menu's category. Afterward, the menu is populated with a small number (around three) of tile selections according to their influence in the current code (Section 3). Each successive selection then causes influences to change and inferences to be made, causing input menus to adapt accordingly. For example, after `Gravity` is selected in Figure 6 for the `My Game` object, the `Ground` and `World Box` traits are promoted to the `My Game` object's root menu since a world with gravity probably needs something to stop objects from falling off the screen!

We have found it useful to create tiles that shape this influence but otherwise lack implementation that drive behavior. For example, selecting `Gravity` in Figure 6 also infers a behavior-free `Side View` trait that only increases the influence of 2D art appropriate for a side view perspective. Likewise, the `Castle Offense` trait shown extended by `My Game` in Figure 6 not only expresses the genre of the game being built, but also improves the influence functionality involved in such games, such as sling shots that "attack" fortifications.

Objects in YinYang are created through library-defined *make* methods, rather than generic `NEW` operators, because otherwise a new object could initially extend anything, and so its input menus would be incomprehensible. A make method can specify a core trait that the object must extend so exclusive extension (Section 3) can filter the input menu accordingly. Make methods can also bind slots in the context of a creating object; e.g., a `World` object can call a `Morph`



(a) `Front` (b) `Fired` (c) `Dequeue`

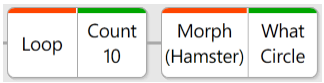


(d)

Figure 7. Input menus for specifying behaviors of a `Hamster` object where (a) the object's code is empty; (b) after the `Front` state is selected in (a); (c) after the `Fired` event is selected in (b); and (d) the object's code after `Dequeue` is selected from (c).

make method to create a new "morph" object, binding this new object's container slot to itself.

Inference, slots, and influence allow simulation behavior to be expressed very rapidly. Consider the input menus in Figure 7 for encoding the behavior of several `Hamster` objects that will be hurled as ammo in the `My Game` object of Figure 6. Initially, these `Hamster` objects are created with the following behavior (code) in the `My Game` object:



Behaviors in YinYang consist of multiple "acts" that evaluate successively horizontally as they "complete." The `Loop` act here completes 10 times causing 10 hamster "morph" objects to be created. The (a) input menu in Figure 7 is used to select the trait of its first behavior, among which the `Front` state tile is shown with prominence because the `My Game` object extends the `Castle Offense` trait. Selecting the `Front` state tile infers, through binding of `Hamster`'s container slot, that the `My Game` object extends a `Queue` trait, causing a queuing area to be created in the world. The `Front` state's own behavior then causes the hamster objects to line up and wait in the world's queuing area, but only completes for the hamster at the "front" of the queue. The `Fired` event is selected from the next (b) input menu, inferring a call to the `Load` method (behavior inference of Section 2) and inferring, again through slot binding, that `My Game` has a sling shot that the player can fire. The `Load` method then causes the "front" hamster to automatically move from the queue to the sling shot, while the `Fired` event then completes when the user has used the sling shot to launch the hamster. Finally, the `Dequeue` action selected in (d) causes the next `Hamster` in the queue to complete its extension of `Front` and move to the sling shot, allowing the player to fire the sling shot again.

Discussion

Although YinYang is very much an early prototype, it is useful in obtaining early experience on our ideas for improving code completion. Informally, we have found that inference clearly makes tablet programming noticeably better than in [19] as selections can be made out of order, which is otherwise a big problem with structured editors. However, we find ourselves making questionable decisions in library design to amplify the usefulness of inference; e.g., we constraints a `Queue` trait for the `World` object that is useful for inferring functionality but basically limits the world to exactly one queue object; what if we want two or three queues? Adding a mechanism to infer member objects and not just traits would allow library design to be more natural.

Although YinYang's editor makes inferences explicit, e.g., `Load` is inferred because of `Fired` in Figure 7 (d), inference still detracts from programmer control and awareness. Inference can infer non-local structure that the programmer cannot immediate see in the editor, while inferences can only be undone by deleting all inferring selections. Inferred exclusive extensions can be particularly annoying as they cause choices to be hidden in input menus, making mistakes more difficult to recover from. Better tooling can help with these problems, but eventually it might make sense to provide programmers with more control over inference, possibly by having them confirm some inferences explicitly.

In contrast to inference, our experience with influence is more mixed. Although we can definitely construct examples that demonstrate its usefulness (as in this section), it is less clear how this feature would scale for real code and libraries. Influence as defined as paths in the extension graph is perhaps too simplistic: we can express "explosives often explodes" and "fired ammo often explodes," but not the conjunction "fired explosive ammo often explodes." On the other hand, influence that is more expressive would necessarily be more complicated. Also, choosing good influence encodings is difficult without data on how the library is

likely to be used; i.e., the "cold start" problem cannot really be avoided by encoding influence relationships by hand. A hybrid approach that combined library designer input with data mining and automated statistical analysis of existing codebases would make influence more usable and useful.

Although we demonstrate inference and influence in a graphical language, we believe that these ideas are transferable to the design of textual general-purpose languages as well, which could also benefit from better code completion. Language-aware text editors could deal with a global namespace by using secondary names to request disambiguation as necessary, or influence could be used to choose a clear "best" choice. Code completion menus could then mirror the structure of YinYang's input menus with more keyboard friendly features such as filtering via partial completions.

5. Technology

Implementing inference and influence involves processing the extension graph as described in Section 3. We present a more thorough discussion of this processing through idealized functional pseudocode that mirrors the semantics of our actual implementation. Without considering slots or binding, a program p consists of trait definitions of the form $\texttt{TRAIT A}\ \overline{\alpha/\omega\ \texttt{B}}$ where A and B are trait symbols, α and β are extension operators selected from Figure 4 in Section 3, and overbar expresses repetition and/or sequencing. Objects in p are encoded as one trait extended by the object, and so we do not consider them here. The program's extension graph $G(p)$ is then defined as:

$$G(p \equiv \overline{\texttt{TRAIT A}\ \overline{\alpha/\omega\ \texttt{B}}}) = \overline{\overline{\texttt{B}\ \alpha\ \texttt{A}}}\ \circ\ \overline{\overline{\texttt{A}\ \omega\ \texttt{B}}}$$

As a notational convenience, $\overline{\texttt{B}\ \alpha\ \texttt{A}}$ expresses that A is duplicated for each element of $\overline{\texttt{B}\ \alpha}$ because A is not grounded by the bottom overbar in the case definition of G as B and α are. Additionally, the circle operator ($\circ$) expresses list concatenation that here collects all the edges of the graph into one sequence. The G function then de-sugars acyclic but multi-directional / extensions into a cyclic graph where each edge is labeled by one operator.

Computing the influence D for a trait A that is or could be extended by another trait B is a shortest path graph problem that can be solved with Dijkstra's Algorithm [8]; $D(p, \texttt{A}, \texttt{B})$ computes the distance of the shortest path in the extension graph from A to B, and hence the influence of B to be extended by A. Likewise, computing a list of traits that could be extended by A according to influence is simply a shortest path traversal of all nodes reachable from A. Paths are however pruned according exclusive extension (Section 2), where first a helper E function computes what an object extends necessarily, explicitly or inferred:

$E(p, \texttt{A}) = \texttt{A} \circ \overline{E(p, \texttt{B})}\ \texttt{IF}\ \overline{\texttt{A : B}} \subset G(p)$
$E(p, \texttt{A}) = \texttt{A} \circ \overline{E(p, \texttt{B})}\ \texttt{IF}\ \overline{\texttt{A :: B}} \subset G(p)$

Conflict detection is then defined as follows:

$$X(p, \texttt{A}, \texttt{B}) = \exists[\texttt{C}, \texttt{D}, \texttt{E}],\ \texttt{D :: C} \in G(p)\ \&\quad \texttt{D} \in E(p, \texttt{A})\ \&$$
$$\texttt{A} \neq \texttt{C}\ \&\ \texttt{E :: C} \in G(p)\ \&\quad \texttt{E} \in E(p, \texttt{B})$$

The X function evaluates to true when A cannot extend B because of a conflict, and therefore paths through B can be pruned. If a programmer is not allowed to create an obvious conflict—the editor prevents them from selecting a conflicting trait—then there will never be a conflict between two traits that the object necessarily extends. Although this property appears merely true by its own definition, its validity is non-trivial when we consider edges synthesized in the extension graph that reflect slot refinements and binding.

Slots

Slots add significant complexity to our semantics as inference and influence propagates across their refinements and bindings; additional edges must then be synthesized in the extension graph to reflect the object graph topology. We first redefine traits to include slot refinements and bindings:

$\texttt{TRAIT A}\ \overline{\alpha/\omega\ \texttt{B}}\ \overline{\texttt{SLOT Q}\ \overline{e\ \alpha'/\omega'\ \texttt{C}}\ \overline{f = g}}$

where Q identifies a slot while e, f, and g are *facet expressions* that are *paths of slots* $\overline{Q}$ that are accessed relative from A; $\overline{e\ \alpha'/\omega'\ \texttt{C}}$ then encodes slot extension refinements while $\overline{f = g}$ encodes slot bindings. The extension graph will now be defined iteratively rather than functionally. The extension graph's initial topology is seeded by the program as follows:

$$G(p \equiv \overline{\texttt{TRAIT A}\ \overline{\alpha/\omega\ \texttt{B}}\ \overline{\texttt{SLOT Q}\ \overline{e\ \alpha'/\omega'\ \texttt{C}}\ \overline{f = g}}}) =$$
$$\overline{\overline{\texttt{B}\ \alpha\ \texttt{A}}}\ \circ\ \overline{\overline{\texttt{A}\ \omega\ \texttt{B}}}\ \circ\ \overline{\overline{\texttt{C}\ \alpha'\ \texttt{A}.e}}\ \circ\ \overline{\overline{\texttt{A}.e\ \omega'\ \texttt{C}}}$$

Bindings have different effects that will be described later. We redefine D to compute a current known shortest path between two traits that is a constraint of the form $\leq\ \phi$. Our first synthesis rule, the *facet rule*, ensures that distance relationships from a facet mirror those of its container:

$$\frac{D(p, \texttt{A.e}, \texttt{B.f}) \leq \phi}{D(p, \texttt{A.e.Q}, \texttt{B.f.Q}) \leq \phi}$$

where e and f are paths of zero or more slots. As an example, consider some code from Section 2:

```
TRAIT Widget { SLOT container */: Panel; }
TRAIT WidgetInCanvas */: Widget
{ REFINE container */: Canvas; }
```

By the facet rule,
$D(p,$ `WidgetInCanvas.container`,
`Widget.container`$) \leq 0$
is true given $D(p,$ `WidgetInCanvas`, `Widget`$) \leq 0$. The facet rule is fairly intuitive and closely resembles how path dependent typing works in Scala [26].

Turning the facet rule around, a less obvious *aside rule* ensures that distance relationships from a container mirror those of its facets:

$$\frac{D(p, \texttt{A.e.Q}, \texttt{B.f.Q}) \leq \phi}{D(p, \texttt{A.e}, \texttt{B.f}) \leq \phi}$$

In our previous example, we stated that if a widget extends `WidgetInCanvas`, then its container must extend `Canvas`. However, if the widget's container extends `Canvas`, should the widget extend `WidgetInCanvas`? Sure, consider:

```
TRAIT WidgetInCanvas */: Widget
{ REFINE container :/: Canvas; }
TRAIT MyWidget { container */: Canvas }
```

Both extension operators in the `container` refinement of `WidgetInCanvas` are now necessary, which by the aside rule, means that whenever a widget's container extends `Canvas`, a zero-distance edge from the widget is synthesized to `WidgetInCanvas`. $D(p, \texttt{MyWidget}, \texttt{WidgetInCanvas}) \leq 0$ is then true by the aside rule because
$D(p, \texttt{MyWidget.container}, \texttt{Canvas}) \leq 0$ and
$D(p, \texttt{Canvas}, \texttt{WidgetInCanvas.container}) \leq 0$ are given.

The aside rule complicates conflict detection that arise from exclusive extensions, where an extension by a facet can create a conflicting extension in its container. Our understanding of this problem is still not well developed, but we reason that the definition of X can be improved to trace more exclusive extension relationships about a node in the extension graph, its facets, and its containers, so that such constructions can be disallowed in a modular way. At the very worst, reverse extension operators can be disallowed from being necessary; they can at least still be very influential (`+++`) that are safely pruned when conflicts occur.

Binding

Binding is similar to extension in that it forms edges in the extension graph that determine shortest path relationships. However, while an extension edge (synthesized or otherwise) simply goes into the extension graphs, bindings must be propagated across extensions. Consider:

```
TRAIT Morph */:: Proto
{ SLOT parent */: Proto;
  REFINE world = parent.world; }
```

This code expresses that the world that a morph exists in is the same as its parent. Given `MyMorph : Morph`, this binding is then replicated for `MyMorph`'s `world` and `parent.world` facets. This replication can be expressed as:

$$\frac{B(p, \texttt{A.e.G}, \texttt{A.f.G}) \leq \phi_b \qquad D(p, \texttt{B.e}, \texttt{A.f}) \leq \phi_d}{B(p, \texttt{B.e.G}, \texttt{B.f.G}) \leq \phi_b + \phi_d}$$

where B represents binding relationships that are initially seeded by bindings encoded in the program:
$B(p \equiv \texttt{TRAIT A} \ldots \overline{\overline{f = g}}, \overline{\texttt{A}.f, \texttt{A}.g}) \leq 0$
$B(p \equiv \texttt{TRAIT A} \ldots \overline{\overline{f = g}}, \overline{\texttt{A}.g, \texttt{A}.f}) \leq 0$
B relationships then generate D relationships that affect extension graph shortest path computations:

$$\frac{B(p, \texttt{e}, \texttt{f}) \leq \phi}{D(p, \texttt{e}, \texttt{f}) \leq \phi}$$

However, binding edges are fundamentally weaker than typical extension edges in one very important way: binding relationships cannot be derived across other binding relationships. Consider the following example:

```
TRAIT Proto *:: Root { SLOT world : World; }
TRAIT World *:: Proto { REFINE world = <>; }
TRAIT Morph *:: Proto { ... }
```

This code expresses that every proto object is related to a world (through the `world` slot) and that the world of a `World` object is itself; the syntax `<>` here is an empty slot path that means the `world` slot is bound to the object that extends the containing trait. Now, if extension edges generated from binding relationships were considered as vanilla D relationships, then we would derive that
$B(p, \texttt{Morph}, \texttt{Morph.world}) \leq 0$ given
$D(p, \texttt{Morph.world}, \texttt{World}) \leq 0$,
$D(p, \texttt{World}, \texttt{World.world}) \leq 0$, and
$B(p, \texttt{World.world}, \texttt{World}) \leq 0$, which clearly leads to a conflict since both `Morph` and `World` exclusively extend `Proto`. However, observe that $D(p, \texttt{World}, \texttt{World.world}) \leq 0$ is derived from a binding relationship; by weakening this relationship to prevent its use in deriving other B relationships, our problem is solved. The fact that D relationships that traverse B relationships are weaker than typical D relationships greatly complicates our semantics: we must now treat distances that go through binding relationships separately from those that do not so that binding propagation can be disabled for those paths that depend on binding.

Miscellaneous and Implementation

Treating methods as extensions requires more changes to our semantics as a method called in code can affect the extension edges of the calling object; e.g. calling the `Pressed` event on a widget causes a necessary edge to be created from the widget to `Button`. We express this by tracking where extension edges go from nodes that are related to methods to those that are not, which are then propagating down to the code that calls the method, which can then be "reaped" into synthetic extension edges by the calling object.

Given its novelty, more work must be done to determine whether YinYang's type system is actually safe and sound, as well as how to interpret these properties with inference. However, we have shown that the type system is at least plausible and can be described concisely. Our implementation of this type system relies on an iterative algorithm to generate synthetic edges that are necessary for shortest path computations: the algorithm propagates facet, aside, and binding relationships along extension edges, which are processed to determine what new edges need to be added from that node. In order for the algorithm to converge and operate efficiently, especially since completion menus must be computed in real time, the following tradeoffs are made:

1. Synthetic edges at or below neutral influence ($* = 64$) are not computed;
2. Slot paths can list a slot at most once; e.g. `World.world.world` is never considered; and
3. A facet is not represented as a node in the extension graph unless required for correctness.

6. Related Work

Code search systems allow programmers to query for code to reuse through specifications, types, patterns, keywords, and so on; see [30] for a survey. Basic keyword-based search engines such as Koders [29] are more apt in finding out how APIs are used, rather than discovering the APIs themselves. Sourcerer [14] moves beyond keywords with more powerful indexing that ranks code ("CodeRank") through machine learning. Reiss [30] demonstrates how programmers can use semantics to find code that they want generated to a form they can use. ProcedureSpace [2] leverages natural language annotations so programmers can find code using natural language queries. The general problem of code search is not addressed in this paper: reusable code of arbitrary granularity could be found in a very large library as abstractions.

Closely related to code search and more related to our work are API search and recommendation systems that help programmers find and use APIs. CodeBroker [36] supports editor-integrated search of a component repository (i.e., a large library) by performing similarity analysis between comments, signatures, and API documentation, which ignores depth and provides ranked results. API Explorer [9] tackles a "discoverability problem," which resembles our depth and breadth problems, to help programmers discover APIs that are not directly reachable from their context by leveraging structural relationships between API elements. PARSEWeb [35] and XSnippet [32] leverage type information to suggest how programmers move from one type to another, basically tackling the problem of depth implicitly. MAPO [37] mines API usage from large repositories to recommend related APIs as well as API usage.

Existing work also focuses specifically on making code completion better. The work in [5] and [31] explores how examples from code repositories and program history respectively can be used to improve code completion through informed ranking. The aptly named BCC (Better Code Completion) [13] explores how enhanced semantic grouping, filtering, and ranking can improve the code completion experience. Little and Miller [15] explore how natural language keywords can "complete" into a statement that possibly calls multiple APIs by mining API documentation in code repositories. API recommender tools and enhanced code completion are limited in solving the depth and breadth problems because they must necessarily work outside of the language's semantics and rely on various heuristics; e.g., similarity or structural analysis. We instead show how language design can solve these problems directly, making such tools easier or even trivial to create. Much of this work is also based on analysis of large amounts of existing code, which is not feasible when designing a new language from scratch.

We were unable to previous work that uses programming language design to specifically make finding things in libraries easier. Component languages such as Scala [26] focus on the safe and expressive reuse of abstractions; but say nothing about how to find these abstractions, which is solely a tooling concern. Our own traits are inspired by Scala's, and our slot design arose from experience using Scala to encode object graphs through path dependent types. Languages that support virtual classes [11, 17] can also perform type-based reasoning, but not inference, over object trees.

7. Conclusion

Programming language designers must stop focusing only on expressing elegant computations and abstractions, and start considering the entire programmer experience; specifically, programmers need help finding things in a large library that often attracted them to a language in the first place. This paper has demonstrated how language design can improve the performance of code completion, providing programmers with better "GPS" when exploring the library. The inference and influence techniques described here can serve as a starting point for a further discussion about how language design can improve library exploration.

As future work, inference can expand to include chains of method calls and property accesses, further broadening what code completion menus can list. Influence can be improved with more expressive models and we should also look beyond type relationships to include probabilistic models mined from example code as well as socially sourced popularity ratings. Looking forward, we should keep improving our languages and tools until we can realize a very large "kitchen sink" library of everything, all easily discoverable through a code completion menu.

Acknowledgments

Thanks to Jonathan Edwards for inspiring this paper's problem and title, and Lidong Zhou for reviewing early drafts.

References

[1] C. Anderson. *The Long Tail: Why the Future of Business Is Selling Less of More*. Hyperion, 2006.

[2] K. C. Arnold and H. Lieberman. Managing ambiguity in programming by finding unambiguous examples. In *Proc. OOPSLA Onward*, pages 877–884, 2010.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. of OOPSLA/ECOOP*, pages 303–311, 1990.

[4] R. A. Brooks. Planning is just a way of avoiding figuring out what to do next. In *MIT Artificial Intelligence Laboratory Working Papers*, September 1987.

[5] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proc. of ESEC/FSE*, pages 213–222, 2009.

[6] G. Curry, L. Baer, D. Lipkie, and B. Lee. Traits: An approach to multiple-inheritance subclassing. In *Proc. of SIGOA*, pages 1–9, June 1982.

[7] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. of POPL*, pages 207–212, 1982.

[8] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[9] E. Duala-Ekoko and M. P. Robillard. Using structure-based recommendations to facilitate discoverability in APIs. In *Proc. of ECOOP*, pages 79–104, July 2011.

[10] J. Edwards. Subtext: uncovering the simplicity of programming. In *Proc. of OOPSLA*, pages 505–518, 2005.

[11] E. Ernst. Family polymorphism. In *Proc. of ECOOP*, pages 303–326, 2001.

[12] W. E. Hick. On the rate of gain of information. *Experimental Psychology*, 4(1):11–26, 1952.

[13] D. Hou and D. M. Pletcher. Towards a better code completion system by api grouping, filtering, and popularity-based ranking. In *Proc. of RSSE*, pages 26–30, 2010.

[14] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.*, 18(2):300–336, Apr. 2009.

[15] G. Little and R. C. Miller. Keyword programming in Java. In *Proc. of ASE*, pages 84–93, 2007.

[16] M. B. MacLaurin. The design of Kodu: a tiny visual programming language for children on the Xbox 360. In *Proc. of POPL*, pages 241–246, January 2011.

[17] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *Proc. of OOPSLA*, pages 397–406, Sept. 1989.

[18] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proc. of PLDI*, pages 48–61, 2005.

[19] S. McDirmid. Coding at the speed of touch. In *Proc. of SPLASH Onward*, pages 61–76, October 2011.

[20] Microsoft Corp. Microsoft announces Visual Basic 5.0, Professional Edition. `www.microsoft.com/presspass/-press/1997/feb97/vb5propr.mspx`, 1997.

[21] Microsoft Corp. Using IntelliSense. `msdn.microsoft.com-/en-us/library/hcw1s69b.aspx`, 2003.

[22] D. P. Miller. The depth/breadth tradeoff in hierarchical computer menus. In *HFES 25th Annual Meeting*, pages 296–300, 1981.

[23] A. Nathan. *Windows Presentation Foundation Unleashed (WPF) (Unleashed)*. Sams, 2006.

[24] L. R. Neal. Cognition-sensitive design and user modeling for syntax-directed editors. In *Proc. of CHI*, pages 99–102, 1987.

[25] K. L. Norman. *The Psychology of Menu Selection: Designing Cognitive Control at the Human/Computer Interface*. 1991.

[26] M. Odersky and M. Zenger. Scalable component abstractions. In *Proc. of OOPSLA*, pages 41–57, 2005.

[27] T. E. Oliphant. Python for scientific computing. *Computing in Science and Eng.*, 9:10–20, May 2007.

[28] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: bringing order to the web, 1999.

[29] B. E. Pangburn and J. P. Ayo. Koders - source code search engine. www.koders.com, 2005.

[30] S. P. Reiss. Semantics-based code search. In *Proc. of ICSE*, pages 243–253, 2009.

[31] R. Robbes and M. Lanza. Improving code completion with program history. *Automated Software Eng.*, 17(2):181–212, June 2010.

[32] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. In *Proc. of OOPSLA*, pages 413–430, 2006.

[33] T. Teitelbaum and T. Reps. The cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24:563–573, September 1981.

[34] B. Templeton. Alice pascal. `www.templetons.com/brad/-alice.html`, 1985.

[35] S. Thummalapenta and T. Xie. PARSEWeb: a programmer assistant for reusing open source code on the web. In *Proc. of ASE*, pages 204–213, 2007.

[36] Y. Ye and G. Fischer. Context-aware browsing of large component repositories. In *Proc. of ASE*, 2001.

[37] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proc. of ECOOP*, pages 318–343, July 2009.