*Journal of Universal Computer Science, vol. 20, no. 2 (2014), 135-168*
*submitted: 1/2/13, accepted: 2/1/14, appeared: 1/2/14 © J.UCS*

# DIE: A Domain Specific Aspect Language for IDE Events

**Johan Fabry, Romain Robbes**
(PLEIAD Lab, Computer Science Department (DCC)
University of Chile, Santiago, Chile
{jfabry / rrobbes}@dcc.uchile.cl)

**Marcus Denker**
(RMoD, INRIA Lille Nord Europe
Lille, France
marcus.denker@inria.fr)

**Abstract:** Integrated development environments (IDEs) have become the primary way to develop software. Besides just using the built-in features, it becomes more and more important to be able to extend the IDE with new features and extensions. Plugin architectures exist, but they show weaknesses related to unanticipated extensions and event handling. In this paper, we argue that a more general solution for extending IDEs is needed. We present and discuss a solution, motivated by a set of concrete examples: a domain specific aspect language for IDE events. In it, join points are events of interest that may trigger the advice in which the behavior of the IDE extension is called. We show how this allows for the development of IDE plugins and demonstrate the advantages over traditional publish/subscribe systems.
**Key Words:** IDE, Plugins, Development Environment, Domain Specific Aspect Languages, Aspects
**Category:** D. 2.3, D. 2.6, D. 3.2

## 1 Introduction and Problem Statement

Over the last decades, software development has moved farther and farther away from simple text editors; the *Integrated Development Environment* (IDE) has become the standard tool for programmers, especially in industrial settings with languages like Java, C# or C++.

Integrated development environments provide additional support to the fundamental edit-compile-run cycle of development and maintenance; these range from basic, state-of-the-practice tools such as syntax highlighting, code autocompletion, or notification of changes, to experimental tools such as task context management [Kersten and Murphy(2006)] and code clone detection and management [Ekoko and Robillard(2008)].

Besides standard integrated tools, it has become clear that the power of IDEs lies especially in extensions developed by third parties. The IDE becomes a platform where innovation is done at the edges, by open source developers, companies and researchers. To support extensibility, modern IDEs such as Eclipse provide a Plugin API [Clayberg and Rubel(2008)].

The amount and diversity of available plugins for Eclipse — there are more than 1,000 plugins in the Eclipse Marketplace[1] — shows that plugins are a powerful way to extend IDEs. Yet, standard plugin solutions have shortcomings.

The first problem is that only functionality exposed by an API can be used by plugins. Functionality that was not anticipated by the developers of the IDE can not be hooked into and thus its use in plugins is impossible. This is because hooks into the IDE cannot be specified as part of the plugin (externally), since they are defined and fixed internally as part of the plugin API.

The second problem is related to the notion of event and notification handling. Plugins hook into the core of the IDE, adding their notification system for the actions taking place inside of the IDE. The events that they track can differ, ranging from events produced by the IDE (changes to the code, navigation in the code, tool usage, copy/paste, etc.) to events produced by the program that is being developed at runtime, as generated by debugging or profiling tools. Currently, these plugins contain an ad-hoc event handler, *i.e.,* each tool implements its own event handlers and event generators. As a result, there is little reuse in the event-handling part of the construction of plug-ins: each developer needs to figure out individually what API to consult, and whether it fulfills his or her requirements. Furthermore, the different event handlers in the system may interfere in each others' operation. Lastly, various event generators may have heterogeneous interfaces as well, so that a tool needing to handle several kinds of events is exposed to additional complexity.

IDEs need to be improved to allow tools to be integrated in a fluid manner, so that plugins can more easily specify how and where they want to extend the IDE's functionality. In addition, we need a flexible and uniform way to couple events happening in the IDE to the functionality provided by the plugin.

The idea we present in this article is a Domain-Specific Aspect Language (DSAL) called DIE. It provides a set of domain-specific abstractions for integrating IDE plugins, in addition to general-purpose Aspect-Oriented (AO) abstractions. Conceptually, the IDE abstractions are a case of event-based Aspect Oriented Programming (AOP) [Douence et al.(2002)]: the IDE generates a stream of events, known as join points, and the correct abstraction level is obtained by a higher-level join point model. The tools express their interest in a certain kind of IDE events by declaring a pointcut on these events. Furthermore, they may also modify, respectively intercept, any other part of the structure or execution of the IDE using general-purpose pointcuts.

A DSAL for IDE events allows tool builders to declare event handlers in a concise and homogeneous way, allowing IDE extensions and tools to express interest in different streams of events seamlessly, with a consistent syntax regardless of the source of the event. We expect that such a DSAL will significantly help

[1] http://marketplace.eclipse.org

tool builders in integrating their extensions with the IDE, and make it easier for them to collaborate with one another.

We first present background material (Section 2) and discuss the properties of a practical system for IDE extensions. We then (Section 3) introduce examples of extensions that we plan to support with our system, and the interactions they need with the IDE they extend. In Section 4, we present our solution — DIE, standing for a DSAL for IDE Event handling —, and how it addresses the examples. We then abstract away from the examples, and describe the solution more formally, followed by an overview of the implementation in Section 5. Abstracting further, we discuss the characteristics of DIE compared to other systems, and assess its expressiveness and generality in Section 6. This is followed in Section 7 by an overview of related work. Finally, we conclude and outline future work in Section 8.

## 2 Background

### 2.1 Aspect-oriented Programming

DIE is an instance of Aspect-oriented programming (AOP) [Kiczales et al.(1997)]. For clarity, we briefly recap the AOP terminology used in this paper here. AOP proposes to modularize *cross-cutting concerns*: concerns the code of which is scattered among different classes in an application. Using AOP these concerns are encapsulated in a new kind of module: an *aspect*. Aspects are a special kind of module because they firstly implement the behavior of a concern, and secondly also specify **when** this behavior should be invoked. The former is called *advice* and the latter are called *pointcuts*. To enable an aspect to work, conceptually each step of the application is reified in what is called a *join point*. The execution of the application consequently produces a stream of join points that is presented to the aspects. The aspects identify the relevant steps in the execution of the application (*i.e.,*the relevant join points), by means of pointcuts that match on the stream of join points. The advice (*i.e.,*the behavior of the aspect), is linked to the pointcut such that when a pointcut matches, the corresponding advice is executed.

Additionally, some aspect-oriented implementations provide for *inter-type declarations* (ITDs): behavioral extensions to given classes. Such declarations can be new instance variables that conceptually belong to the class, and not to the aspect (to increase modularity), and additional behavior (*i.e.,*new methods that, for the same modularity concerns, are added to the advised class as well). Such features are also present in Traits [Ducasse et al.(2006)], however the latter do not possess the join point - advice mechanism.

Seminal work on Aspect-Oriented Programming (AOP) used aspect languages that were specific to the concern of the aspect [Cleenewerck et al.(2008)].

Put differently, these aspect languages were in fact Domain-Specific Languages (DSL) [van Deursen et al.(2000), Mernik et al.(2005)]. To the best of our knowledge, all definitions of what a DSL is are however somewhat open to interpretation. A recent definition of DSL is given by Fowler and Parsons, as follows: "**Domain-specific language** (noun): a computer programming language of limited expressiveness focused on a particular domain." [Fowler and Parsons(2011)] DSLs are also known as little languages: programming languages that are specifically designed to express applications in a particular domain. In general, language constructs of a DSL directly reflect the concepts of the domain and hide the DSL programmer from non-domain-specific technical issues. As a result, specifications written in a DSL are more declarative, more abstract, and are closer to the intended behavior [van Deursen et al.(2000), Mernik et al.(2005)]. They have been shown to significantly increase the correctness and speed of comprehension of code [Kosar et al.(2012)], which in turn should ease application development and maintenance. To emphasize their aspectual nature, DSLs for the description of cross-cutting concerns are also known as Domain-Specific Aspect Languages (DSAL).

DIE is a DSAL; In the case of DIE, conceptually the events of the IDE are reified as join points that are domain-specific, and each execution step of the IDE as general-purpose join points. The aspects written in DIE match on the stream of join points by specifying pointcuts, *i.e.,*the pointcut specifies how the plugin hooks into the operation of the IDE. The advice of the plugin then provides the implementation of the functionality of the plugin. Further, we will see that DIE introduces—although in a preliminary form— user interface (UI) ITDs; DIE sees the widget structure that describes the IDE and may add extra widgets there, when needed by a plugin.

### 2.2 Publish/Subscribe systems

As DIE defines events that happen in the IDE and provides a way to bind those events to actions, it is tempting to see DIE as equivalent to a standard Publish/Subscribe system. We argue here that this is not the case as the event based AOP foundations provide some clear benefits over Publish/Subscribe. This benefits are: unforeseen flexibility, behavior extensibility, user interface extensibility, and performance.

#### 2.2.1 Unforeseen flexibility

Event based systems provide a fixed set of events. New events can only be introduced with changes to the code of the publisher (the IDE). Plugins (the subscribers of events) can not extend or change the events defined by the publisher. With DIE, we can define pointcuts based on the event-based join points, which is similar to subscribing to events. But contrary to Publish/Subscribe, the developer is free to define externally new events as needed (as shown in Section 4.4). This addresses the rigidity of event-based systems: if there is no domain-specific event representing the behavior of interest, we can simply fall-back on pointcuts on the stream of general-purpose join points to capture it.

### 2.2.2 Behavior extensibility

In addition to extending the event model from the outside, with DIE we can go beyond a simple event model and use the full power of AOP. For instance, control flow pointcuts allow for filtering out of events depending on their run-time context. With ITDs we can add state or behavior if needed. One example; the history plugin (in Section 4.4), shows how these advanced features are used in combination with the domain-specific pointcuts and simplify the implementation. In the example, if one were to store the navigation history outside the browser class (without using ITDs), then one has to ensure that it is garbage-collected properly once the associated browser is closed, making the implementation more complex.

### 2.2.3 User interface extensibility

DIE offers the ability to provide ITDs at the user interface (UI) level; this is crucial when one needs to expose the new functionality of the plugin to the user. Most UI frameworks implicitly order widgets as a tree; DIE provides primitives to insert new UI elements in existing graphical widget layouts, by referencing a select type of IDE windows and permitting the insertion of new UI widgets relative to in those windows. This is only a preliminary version of user interface extensibility. The concept is straightforwardly extensible to inserting new UI elements in existing graphical widgets layouts, by referencing existing widgets (through a pointcut), and permitting the insertion of new UI widgets relative to these widgets.

### 2.2.4 Performance

The creation of events can easily lead to performance problems, as the event creation code is always executed. With AOP in general, and specifically in DIE, we instead only create join points (events) when needed, with as few context information as needed, following well-known techniques [Hilsdale and Hugunin(2004), Masuhara et al.(2003)]. This especially means that defining an event type does not have any performance impact if these events are currently not being consumed.

## 3 Motivating Examples

To illustrate the benefits of using DIE, we now describe several examples of tools that need a non-trivial integration with the IDE. In the next section we will show how DIE enables this integration, in an uniform fashion.

### 3.1 Recent Changes

It has been shown that programmers often come back to classes and methods they changed recently in order to change them again [Robbes et al.(2010)]. A simple way to improve productivity is to exploit this programmer behavior, and provide an on-demand list of recently changed method and classes. Such a plugin needs the following information from the IDE:

- Notification of recent changes: Each time the developer changes a method, it needs to be put in first position in the list of recently-changed methods.
- Keyboard shortcut: To really improve the speed of navigation, the tool must be accessed quickly. The most efficient way is through a keyboard shortcut; the tool should either register itself in a keyboard shortcut database, or monitor keyboard activity itself.
- Once a method or class is selected, the tool requests the IDE to browse it.

### 3.2 Code Completion

Code completion is one of the features most used by programmers in IDEs [Murphy et al.(2006)]. In Pharo Smalltalk [Black et al.(2009], the code-completion algorithm takes into account recent change information to rank matches and also provides an "always-on" interface when the code completion is automatically invoked as the programmer types, rather than explicitly via a keyboard shortcut [Robbes and Lanza(2010)]. Such an interface needs a greater amount of information from the IDE:

- Notification of recent changes: Each time a method changes, the set of recently-used method names is updated.
- Keyboard activity: The code completion window can potentially be invoked after any character is entered in a source code area.
- Focus activity: If a programmer leaves a method, the completion menu, if open, should disappear; upon returning to said method, it should reappear, in the state it was previously.

### 3.3 Window Positioning

Several IDEs feature multiple windows, *e.g.,* in Smalltalk [Goldberg(1984)], but also newer offerings such as Code Bubbles [Bragdon et al.(2010b)]. In these IDEs new windows are often opened and closed; positioning the new window at a sensible location on the screen may become an issue. For instance, if a programmer issues a query, such as asking for the list of users of a method he or she is currently browsing, it is natural to want to position the newly-opened window near to the window displaying the method instead of at an arbitrary position on screen. To do this, a window-positioning system needs:

– Notification of opening windows: So that the default position of the new window can be altered.

– Knowledge of the window triggering the command: So that the window positioning extension knows the current position and size of the window that it should position another window close to.

### 3.4 Browsing History

Navigating the source code is one of the most common activities during program comprehension. Often, programmers navigate back and forth between program entities [Robbes et al.(2010)], as regular users do when they browse the web. However, most IDEs lack the support for browsing previously inspected code fragments; developers have to fall back to manually navigate to these fragments instead. Implementing a browsing history for a source code browser needs several extensions:

– A mechanism to add new navigation buttons and shortcuts to UI elements in a non-intrusive way.

– Keeping track of the navigation activity. In some cases, this functionality is not exposed by the IDE, as it was not envisioned that tools would be interested in it.

– Storing the recent navigation data and binding it to a given code browser, in order to keep the navigation history of each browser separate.

## 4 DIE by Example

To enable tool builders to declare event handlers in a concise and homogeneous way, while maintaining their tool as one separate module, we developed DIE: a DSAL for IDE Event handling. DIE is implemented in Pharo Smalltalk [Black et al.(2009], using PHANtom [Fabry and Galdames(2011), Fabry and Galdames(2013)]: an existing aspect language for Pharo, as a provider of the aspectual functionality.

DIE is a domain-specific aspect language because it provides a unified event notification mechanism that reifies events at a higher conceptual level as domain-specific join points. The DIE join points are domain-specific as their join point types correspond to elements of the domain of IDE's, *e.g.,* an edit action or a method compilation, and the properties they hold are of a domain-specific nature as well, *e.g.,* the nature of the edit action or the method being compiled. DIE is however not limited to this event model, it also allows arbitrary cross-cutting functionality to be expressed in terms of (general-purpose) PHANtom language constructs. As a result, it overcomes one of the limits of publish/subscribe systems: when using these systems, tool builders can only rely on the events present in the event model. Hence DIE is more flexible and more powerful, as discussed in more detail in Section 6.

In this section we present DIE by showing how it enables the examples we discussed previously to be straightforwardly implemented. In each of the examples we focus on how the tool is connected to the event flow as well as to the lower-level actions performed by the IDE, if any. Installation and uninstallation of plugins that use DIE is seamlessly taken care of by PHANtom, as it provides for dynamic installing and uninstalling of aspects at runtime. Also, the core logic of each tool would fundamentally be the same as that of existing tools that provide this functionality. Therefore we do not describe the plugin implementation in this paper. Instead, for each example we briefly restate the needs of each tool, and then describe the DIE pointcuts necessary for its implementation as well as the DIE advice that matches these pointcuts. Note that the syntax of DIE is standard Smalltalk as DIE is an embedded DSL [Hudak(1996)]: it exploits the Smalltalk syntax to express the domain-specific code denoted in its programs. The grammar of DIE is provided in Section 5.5.

### 4.1 Recent Changes

We start with the most straightforward tool, to introduce how our system operates as well as the syntax of DIE.

#### 4.1.1 Needs

In the case of the recent changes tool there are two needs: (1) after methods are compiled, they need to be added to the list of recent methods; and (2) a shortcut key is needed to trigger the UI of the tool.

#### 4.1.2 Registering new methods

This is achieved by the following pointcut and advice:

```
DIEAdvice
    after: MethodCompile
    do: [:result :parent |
        self recentMethods add: (result -> parent). self ui update].
```

*Pointcut:* The first line of the above code defines the pointcut: MethodCompile, and determines that the advice needs to run after the matched join points: DIEAdvice after:.

*Advice:* The second line identifies the actions of the advice with the do: keyword, followed by a Smalltalk block (the remainder of the code). As the IDE for which these plugins are written is Smalltalk and the majority of their implementation will be in Smalltalk, it is an obvious choice for the behavior of the advice to be written in Smalltalk, using blocks (which are equivalent to anonymous functions). Selected pieces of context information of the join point are accessible from within the block. This information is added to the join point by DIE's infrastructure. To use context information, the block declares arguments, in the example this is the first line of code of the block. Each argument must have one of the following names: result, parent or window. This name determines what piece of context information will be bound to that argument when the block is executed. In the example, the argument result contains the result of compiling the method, *i.e.,* the method being added to the system, and parent contains the class to which this method is added. The body of the block simply adds this information to the list of recent methods and then instructs the UI to update itself to reflect this change, if needed. (In Smalltalk the reserved word self refers to the current object, *e.g.,* as the this keyword in Java.)

### 4.1.3 Keyboard shortcut

In order to show the User Interface to the user, a second combination of pointcut and advice implements opening the UI when its shortcut key is pressed.

```
DIEAdvice
    after: ShortcutKey result = [self shortcutKey]
    do: [self userInterface show].
```

*Pointcut:* The pointcut must not simply match the pressing of a shortcut key, it also must establish if the key pressed is the key associated with this tool. To achieve this, the result of pressing the ShortcutKey, *i.e.,* the key pressed, is compared (=) to the shortcut key associated with the tool. To allow any computation to be used in such comparisons while keeping expressiveness, DIE adjusts its behavior when the right hand side of = contains a Smalltalk block; this block is evaluated at each comparison, obtaining the value to compare the left hand side with. In this case, the shortcut key is obtained by sending an accessor message to self; this allows to potentially change the key-binding to better suit the user's preferences.

*Advice:* The body of the advice is straightforward, showing the UI whenever the shortcut key has been pressed. Note that as it does not use any properties of the join point, the block does not need to declare any parameters. This tailoring is an advantage of a DSAL that increases the conciseness of DIE's solution.

## 4.2 Code Completion

### 4.2.1 Needs

The code completion algorithm also maintains a list of recently compiled methods. It contains a pointcut and advice similar to the first example above, which we do not repeat here for brevity. Instead we focus on (1) detecting keyboard activity and (2) detecting focus activity.

### 4.2.2 Keyboard activity

Whenever the user is entering text that is source code, the code completion algorithm must be informed such that it can open its UI if it deems it appropriate. This is achieved by the pointcut and advice below.

```
DIEAdvice
    after: Edit parent isCode
    do: [:parent | self completionAlgo codeEditOn: parent].
```

*Pointcut:* The pointcut first establishes whether the join point is an Edit action on some text, either by a key press or a cut/paste action. If the user interface element that contains the edit action (parent) is an editor of source code (isCode), the pointcut matches.

*Advice:* The advice also obtains the text field where the edit action took place using parent. The text field is then passed to the completion algorithm that decides what action to take based on the contents of it (which includes the location of the cursor, and access to the source code in it).

### 4.2.3 Focus activity

To implement the focus behavior we need two pointcut advice pairs: one to hide the completion UI if open, and one to reopen it if it was previously hidden.

```
DIEAdvice
    before: FocusGain
    do: [self completionAlgo hideUI].
```

*Pointcut:* The pointcut is triggered when a new window gains focus. Any code completion UI in any other window should hide itself just before the new window gains focus via the FocusGain pointcut.

*Advice:* The advice hence informs the completion UI that it should hide itself, and as such is a simple method call. If the completion UI is not opened, it can simply ignore the hide command.

```
DIEAdvice
    after: FocusGain result isCode
    do: [:result | self completionAlgo showUIFor: result].
```

*Pointcut:* The second pointcut advice pair is triggered after a window gains focus, in the case that the widget with the focus contains source code—as the completion is only functional in this case. The pointcut hence matches only text fields that contain text that is considered source code.

*Advice:* The advice informs the completion UI, that if it was open previously, it should now show itself again. The text field is obtained from the join point by asking for the result of the focus gain event.

### 4.3 Window Positioning

The next example shows how one implements window positioning, for browser windows that are spawned by other windows.

#### 4.3.1 Needs

In the Pharo IDE, spawning browser windows happens when a programmer wishes to know where a method is used (senders of a message), to consult all the implementations of the message (implementors), or the class hierarchy for a given class. Intelligent window positioning places such a browser near the originating window. We only give the code for the one DIE advice: opening a senders browser. The implementation for other types of windows is almost identical: the difference is in the identification of the type of window whose opening is intercepted.

#### 4.3.2 Changing window position

Using DIE, altering the position of a window as it opens is done like so:

```
DIEAdvice
    after: WindowOpen result = SendersBrowser
    do: [:result :parent | self move: result over: parent].
```

*Pointcut:* The pointcut of the window positioning tool matches on window open events (WindowOpen) when the window being opened (result) is a senders browser; this context information being added to the join point by DIE. In this comparison the SendersBrowser is used instead of a Smalltalk block. This is possible, since DIE has built-in, uniform comparison operations for all standard IDE windows. Hence, for other windows, *e.g.,* a Hierarchy Browser, only the comparison needs to be changed, in the example to: = HierarchyBrowser.

*Advice:* The advice behavior is specified to occur after the window has opened. It obtains the result of the event, *i.e.,* the opened window, and instructs it to move itself over the window that had focus (parent) when the operation to open the browser was performed. This again illustrates the use of multiple context information elements present in the join point.

### 4.3.3 Doing it Without DIE

As an illustration of the advantages of DIE, we now show an implementation of the same behavior without using DIE. The code below represents our best effort to yield compact yet readable code, maximally exploiting our detailed knowledge of the inner workings of the Pharo Smalltalk IDE.

As a first step, the plugin needs to hook into the IDE to intercept the opening of windows. However, the Pharo Smalltalk IDE (version 1.3) does not define a hook for this. Consequently, the plugin needs to modify the IDE to add code that notifies it to the relevant method in the system. For example this can be performed by replacing the existing openInWorld method of the SystemWindow class by the code below.

```
SystemWindow>>openInWorld: aWorld
   self bounds: (RealEstateAgent initialFrameFor: self world: aWorld).
   WindowPositioningPlugin instance notifyOpen: self. "<– Plugin notification code added"
   ↑self openAsIsIn: aWorld.
```

While performing the above is straightforward to do in Smalltalk, it does couple the plugin to this specific version of the IDE. Also, if any subclass overrides this method and does not call super, these changes will be lost and hence no notification will be performed. Lastly, if we consider other plugins wishing to be notified of windows opening, it does not compose. The different plugins cannot simply change the method to their own version, as installing a plugin would then mean that the notification behavior of an already installed plugin would be lost.

The WindowPositioningPlugin class uses the Singleton design pattern to ensure that there is only one instance of the plugin running, and also provides straightforward access to this instance via the instance class method. For the sake of brevity, we do not include the code for this pattern implementation here as it is trivial.

On notification of the window being opened, the plugin should first establish if the window being opened is a senders browser. For example this can be done by verifying that the model of the window being opened is an instance of the OBSendersBrowser class, as below.

```
WindowPositioningPlugin>>notifyOpen: aSystemWindow
   (aSystemWindow model class = OBSendersBrowser) ifTrue: [self doMove: aSystemWindow].
```

Verifications for other kinds of window could be performed similarly, adding further tests to the class of the model as extra statements in the above method and calling the doMove: method if this is the case. The actual moving of the window would then be implemented in this method, which obtains the top window and then calls the move:over: method, which performs the actual moving. Note that, as above, we obviate the implementation of the move:over: method as it is not relevant to the discussion.

```
WindowPositioningPlugin>>doMove: aSystemWindow
  self move: aSystemWindow over: SystemWindow topWindow.
```

The working of the above method relies on a subtlety of how the notification of window opening is performed. The plugin is notified just between when its position on screen has been calculated (the first line of code in the openInWorld: method above), and it is actually opened (the last line in the same method). As a first consequence of this, the window whose action caused the new window to be opened, *i.e.,*the parent window, is still the topmost window in the IDE. Hence it can be straightforwardly obtained through the SystemWindow topwindow expression. As a second consequence, we can set the final position of the opening window in the move:over: method and be sure that it will not be changed later in the opening process.

The code above requires detailed knowledge of the IDE internals and is much more intricate than the single statement of DIE code given in the beginning of this section. As it requires detailed knowledge of the IDE, it entails more effort to build, or needs to be written by an expert on the IDE itself. Since the amount of code is larger than the DIE code and more intricate, the known advantages of using a DSL for conciseness and readability are lost. Contrast this with DIE, where the event handlers are defined in a concise and homogenous way, without requiring any knowledge of the internals of the IDE.

Moreover the code above is tightly coupled to the exact version of the IDE, where the event notification hook can be placed in this precise moment of window opening. If the window opening logic changes, the plugin would need to be rewritten to change its event notification hook, which can become complicated if bounds are calculated after the parent window is no longer the top window of the IDE. The same would hold for any other plugin that relies on this window opening logic. As DIE provides an infrastructure that abstracts away these implementation details, a change in the IDE implies a change only in DIE, and the plugins using DIE would keep functioning without changes. Lastly, the code above is not composable with regard to multiple plugins wishing to hook into the window opening logic, nor with window subclasses providing a different implementation of this method. This as we simply change the window opening method by a new method that adds a notification. With DIE, the original method is left untouched. Instead the aspect-oriented infrastructure takes care of calling multiple advice, if needed, as well as ensuring that overriding methods of subclasses also trigger a notification.

### 4.4 Browsing History

The last plugin we detail is the most complex so far; it introduces three additional concepts of DIE: use of PHANtom pointcuts, use of PHANtom ITDs, and domain-specific ITDs, also known as UI advice.

#### 4.4.1 Needs

To be able to browse through the history of previously visited methods, several extensions must be done: (1) the browser needs additional state to keep the history information; (2) the browser needs additional behavior to track the methods it browses; (3) the user navigation should be recorded (excluding history usage itself!) and (4) the browser's UI should be extended with new buttons to expose the functionality to the user.

#### 4.4.2 Adding state for history

Before tracking the navigation history, each browser window must first maintain a list of these methods. We can achieve this by using ITDs—the habitual AOP modular class extensions mechanism. The advantage of using ITDs is that these class extensions remain part of the module that declares them, in this case the browsing history plugin. As a result, this plugin remains a fully self-contained module.

For this, we fallback on DIE's aspectual implementation layer, PHANtom. The PHANtom language provides for such class extensions in the following way:

```
PhClassModifier on: Browser addIV: 'history'.
```

The above declares that all instances of Browser, *i.e.,* the code browser windows, now have an extra instance variable named history[2].

#### 4.4.3 Adding user tracking behavior

By using inter-type declarations we can also add behavior to the browser window. In this case we add methods for showing the next and previously visited methods, along with a method that adds the currently visited method to the list of methods (we omit the method bodies: these involve manipulating the list of visited locations, and changing the location the Browser is currently visiting—altogether trivial operations):

```
PhClassModifier on: Browser addIM: 'showNext ...'.
PhClassModifier on: Browser addIM: 'showPrev ...'.
PhClassModifier on: Browser addIM: 'recordVisit ...'.
```

[2] Note that Smalltalk class extensions, by themselves, do not allow instance variables to be added.

#### 4.4.4 Recording navigation

The next piece of the puzzle is registering the navigation to a method in the browser: we need to add this method visit to the history of visits. We should however avoid recording methods being shown as a result of going through the history itself. This is obtained using the code below.

```
DIEAdvice after: (TextFieldOpen result isCode) &
                 (TextFieldOpen window = Browser) &
                 [(PhPointcut receivers: 'Browser' selectors: #(showNext showPrev)) cflowNot]
             do: [:window | window recordVisit]
```

*Pointcut:* This pointcut is more complex than previous examples, as it needs to recognize when history navigation takes place; this is an instance of unforeseen extensibility. The most natural way to do so is to use DIE's ability to use a PHANtom pointcut. PHANtom pointcuts can be control flow-sensitive and as such be triggered only when a method is (or is not) being executed in the control flow of another method.

In this example, the pointcut is a composite pointcut (using the conjunction operator &); the last part of the conjunction is a PHANtom pointcut that uses the negation of a cflow of the showNext and showPrev methods. The concept of cflow—short for control flow—is common in AOP: it essentially identifies all join points that occur due to the execution of (a collection of) methods, *i.e.,*the current control flow of the program. In this case, the pointcut expresses that we want to ignore all events that occur due to the showNext and showPrev methods being executed. Note that functionality that relates various points in the execution of the program is a strong point of AOP that is absent in publish/subscribe systems.

*Advice:* Thanks to the state and behavior extensions made above, recording the actual navigation is a simple task that we do not discuss in detail.

#### 4.4.5 Displaying functionality in the UI

The browser window however lacks specific buttons to navigate to the previous and next entry in the list.

We can add these buttons to the window by capturing the specific part of window construction where the tree of widgets of the window is built. At this point we can add these buttons to the tree, which ensures that they are present when the window is shown. We will not perform this through low-level general-purpose pointcuts, advice and ITDs, but use a higher level of abstraction. We use the domain-specific ITD feature of DIE called UI advice. This allows us to express adding buttons in a concise way, as exemplified by the code to add both "previous" and "next" buttons, below, the result of which is shown in Figure 1.

```
DIEUIAdvice
```

```
    on: Browser
    addButton: '<<' onClick: [:window | window showPrev].

DIEUIAdvice
    on: Browser
    addButton: '>>' onClick: [:window | window showNext].
```

*Pointcut:* We see that the pointcut syntax for UI elements is simple. One only needs to specify the window to modify: in this case a Browser. In the Browser, DIE will add UI elements to the button bar already shown by the Browser.

*Advice:* Each advice inserts its new UI element, in this case a button: one calling the method showNext that goes forward in the history, and the other doing the opposite by calling showPrev. Behind the scenes, DIE ensures that the buttons are inserted after every window refresh.

The UI Advice feature is, for now, very simple: we need to provide an `addButton:` implementation and access to the button bar for all IDE windows where extension is planned. Similarly, this could be extended to also allow the inclusion of other UI elements such as drop down lists and text fields.

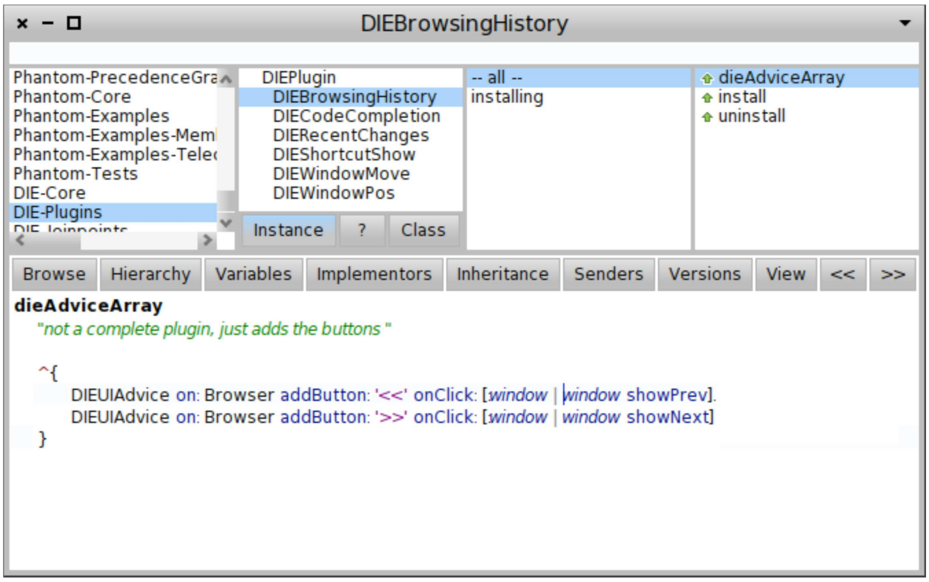


Figure 1: A DIE UI Advice inserting two buttons and the result: the last two on the right of the button bar.

## 5 DIE: design, specification and implementation

### 5.1 DIE design decisions

Rebernak et al. have proposed a list of seven design decisions that should be made when a DSAL is created [Rebernak et al.(2009)]. We use the questions here to document the major design decisions made in DIE.

1. *What are the join points that will be captured in the DSAL?* The join points are the IDE events which are listed in Figure 2, extended with any general-purpose join points that can be captured by PHANtom, and the UI pointcuts outlined in Section 5.4.

2. *Are the DSAL join points Static or Dynamic?* The UI join points are static, all the other join points are dynamic.

3. *What granularity is required for these join points?* The granularity of a join point corresponds to the event that it represents, the corresponding PHANtom join point, or the UI element that is inserted, respectively.

4. *What is an appropriate pointcut language to describe these join points?* The pointcut language of DIE is described in Section 5, *e.g.,*the grammar for pointcuts is given in Section 5.5.

5. *What is advice in this domain?* Advice can be any action that needs to be performed by an IDE plugin, hence the language in which it is specified is a general-purpose language.

6. *Is extension/refinement only about behavior, or also structure?* Extension and refinement is both about behavior (Figure 2 and PHANtom pointcuts) and structure (UI pointcuts of Section 5.4).

7. *How is information exchanged between join points and advice? Is advice parametrization needed?* Information is exchanged via parametrization of the advice, the name of the parameters define what information passes from the join points to the advice, as defined in Figure 2.

### 5.2 The Domain-Specific Join Points of DIE

From the examples we described, we can see that the DIE event system places domain-specific information in each join point, such that the pointcuts and advice can use this contextual information whenever it is needed. In aspect-oriented software development, the choice of what events to reify as join points and which information is contained in a join point is called the join point model. The choice

<table>
<tr><th>Join Point</th><th>result</th><th>parent</th><th>window</th></tr>
<tr><td>FocusGain</td><td>the widget</td><td rowspan="10">User interface element containing the widget or action</td><td rowspan="10">Window of the widget or action</td></tr>
<tr><td>ShortcutKey</td><td>the key pressed</td></tr>
<tr><td>ButtonClick</td><td>button clicked</td></tr>
<tr><td>MenuSelect</td><td>the menu item</td></tr>
<tr><td>ListOpen</td><td>the list widget</td></tr>
<tr><td>ListClose</td><td>the list widget</td></tr>
<tr><td>ListSelect</td><td>old and new sel.</td></tr>
<tr><td>TextFieldOpen</td><td>the text field</td></tr>
<tr><td>TextFieldClose</td><td>the text field</td></tr>
<tr><td>Edit</td><td>edit action</td></tr>
<tr><td>WindowOpen</td><td>the window</td><td colspan="2">previous window</td></tr>
<tr><td>MethodCompile</td><td>the method</td><td>the class</td><td>void</td></tr>
</table>

**Figure 2:** Domain-Specific join points and their properties

that we made for DIE is to take the general-purpose join point model of PHANtom, and add a collection of additional join points, the domain-specific join points of IDE events. The former allows for any execution step of the IDE to be used by the tool builder. The latter allows easy use of events of special interest to tool builders.

Figure 2 shows the domain-specific join points that are provided by DIE. We see that these mainly feature UI events, which is not surprising as we have seen in the examples that the plugins mostly deal with user interactions.

### 5.3 Semantics Overview

DIE being built on top of PHANtom, unsurprinsingly, the semantics of DIE have much in common with the ones of its host language PHANtom. With DIE advice being a block of Smalltalk code (which is also possible in PHANtom), the most significant difference between PHANtom and DIE are the domain-specific pointcuts, and UI pointcuts and advice. Hence in this section, we only highlight the elements unique or particularly relevant to DIE.

#### 5.3.1 Domain-Specific Pointcuts

DIE allows a range of pointcuts from the basic to the most complex. Pointcut expressions in DIE can simply be the name of the join point, as described in Figure 2. For these pointcuts, DIE provides a consistent nomenclature, *e.g.,* it provides definitions for the most common IDE events and entities, such as browser windows or text fields.

### 5.3.2 Falling back to PHANtom

If the domain-specific pointcuts are insufficient, the fallback is provided in the use of general-purpose PHANtom pointcuts. This guarantees even more flexibility in the queries that can be made (such as referencing the control flow). The only limitation to the PHANtom pointcuts is that they cannot reference runtime context information as DIE is not there to provide it.

### 5.3.3 Pointcut context checks

Domain-Specific Pointcuts can also be refined by additional tests on the properties of the joint point. This is performed via accessing the result, parent, and window properties of the join points (defined in Figure 2), and additional verification on these properties. Verifications are either an isCode check or a comparison operator.

Considering the former, the query checks whether the user interface element that is queried is an editor of source code. For example, a pointcut that matches a TextFieldOpen join point if its result fulfills the isCode property, matches on the opening of a text field that is ment to source code.

Considering the latter, the comparison operator allows for matching of a window of a specific kind (*e.g.,*a senders browser), or arbitrary code to be executed in a block. Such a block receives the value to compare to as an argument and may, as shown in the examples, reference arbitrary methods in the aspect itself or elsewhere, in order to allow for maximum flexibility.

### 5.3.4 Context information on demand

DIE's join points allows for conciseness and potential implementation efficiency as the context information needed by advice is provided on an as-needed basis. This is thanks to DIE's advice definitions: referencing a piece of context as an argument to the advice closure is enough to obtain it, while omitting the context means that it can be potentially optimized out.

### 5.3.5 Dynamicity

A characteristic shared with PHANtom is that DIE's aspects are dynamic: they can be deployed or disabled at will. This is essential in the context of IDE plugins as these may need to be enabled or disabled dynamically. One limitation here is that alteration to browers windows are static: only new windows will reflect the fact that a plugin is enabled or disabled at the moment.

#### 5.3.6 Adding state or behavior

ITDs are not unique to DIE or PHANtom. However, they are also important in the context of developing IDE plugins, as the most convenient way to implement a plugin may be to extend existing classes with additional state or behavior. The UI pointcuts and advice described below is a further example of ITDs.

### 5.4 UI pointcuts and advice in DIE

In our iterative definition of DIE, we realized that exposing functionality back to the user is as important as monitoring the events that trigger plugin usage: both concepts intuitively deal with user interaction, and as such they both deserve to be located in the same conceptual implementation layer. This is obvious for instance when one defines keyboard shortcuts to expose the plugin's interface—Keyboard shortcuts are already supported by DIE—; other ways to expose functionality, through UI elements, need to be defined as well.

To introduce other UI elements, we are in the process of extending DIE to support UI manipulation. We already defined domain-specific ITDs for the user interface, which we called UI Advice. UI Advice uses UI pointcuts (where join points are UI elements of interest), while its advice defines operations that manipulate UI elements. In the following, we present our ongoing solution.

#### 5.4.1 UI pointcuts

Our UI pointcuts are basic: only entire windows are supported, as shown in the browsing history example. More complex queries could take advantage of the organization of UI widgets in a tree, much like a program is an Abstract Syntax Tree. To further the comparison, widgets of interest in a UI tree (buttons, lists, panels), have types and names, allowing one to easily write queries matching a subset of all the widgets in a window (*e.g.,* all the widgets of type lists, or the button with the label 'next').

#### 5.4.2 UI Advice

Our UI advice are also very simple, covering the basic case of adding buttons to a dedicated button bar of a window. Other type of advice are possible: hiding existing widgets (enable/disabling a plugin), inserting arbitrary widgets, or modifying properties of widgets (*e.g.,* specifying entities of interest by changing their background color).

#### 5.4.3 Going further

To provide a complete implementation of UI pointcuts and advice beyond our current proof of concept, we need to: (1) formalize the grammar and syntax of use, beyond our initial examples; and (2) abstract from the accidental complexities of Pharo's UI framework, Morphic (ignoring non-significant widgets, such as layout widgets; or handling idiosyncrasies related to the proper update of some widgets). Both directions are promising areas of future work.

### 5.5 The grammar of DIE

As the main contribution of DIE lies in the domain-specific join points of Figure 2, its language constructs mainly concentrate on matching on these join points and extracting context information. The grammar of DIE is given in Figure 3, and we discuss its main features next[3]:

- A DIE pointcut (prim-pointcut) consists of the name of the join point, as in Figure 2, optionally followed by a context check.
- Context checks (context-check) consist in the name of the context property, as in Figure 2, followed by an equals operation or an isCode test.
- The right hand side of an equals operation can be the name of a standard IDE window, a zero-argument block to test object identity, or a one-argument block that defines a custom equality operator.
- Pointcuts that have a context check can be combined with & and |, for conjunction and disjunction respectively.
- Instead of a DIE pointcut, a PHANtom pointcut may be used. This however does not cause the context information to be filled. Hence these should only be used as part of a conjunction or disjunction.
- Advice of DIE is a Smalltalk block that can extract context information of the join point by specifying arguments whose name is the context property.

### 5.6 DIE implementation

DIE is an embedded or internal DSL [Hudak(1996)]: it adds domain-specific pointcut-advice pairs to PHANtom, exploiting the Smalltalk syntax to do so. The implementation of DIE mainly consists in translating DIE statements to the equivalent pointcut-advice pairs of PHANtom, as outlined below. This implementation has multiple advantages of which we highlight three here: Firstly

[3] This grammar does not contain UI advice, as its definition is not stable enough yet to be frozen in documentation

```
program       = 'DIEAdvice', kind, pointcut, 'do:', advice, '.';
kind          = 'before:'
              | 'after:';
pointcut      = ('(', pointcut , ')')
              | pointcut-comb
              | phpointcut
              | prim-pointcut;
pointcut-comb = (pointcut , '&' , pointcut)
              | (pointcut , '|' , pointcut)
              | (pointcut 'not');
phpointcut    = '[' , Smalltalk-statements , ']';
                    (* should return a Phantom pointcut *)
prim-pointcut = join-point-name , [context-check];
context-check = property-name , ('isCode'
                                | '=' , compared );
property-name = result
              | parent
              | window;
compared      = IDE-window
              | 0-arg-Smalltalk-block
              | 1-arg-Smalltalk-block;
advice        = 0-arg-Smalltalk-block
              | special-block;
special-block = '[' , (':result' | ':parent' | ':window'),
                    [(':result' | ':parent' | ':window')],
                    [(':result' | ':parent' | ':window')],
                         (* each property may only appear once *)
                    '|' , Smalltalk-statements , ']'
```

Figure 3: The grammar of DIE in EBNF. Allowance for white space is omitted and the nonterminals that contain the word Smalltalk correspond to Smalltalk grammar entities.

it inherits from PHANtom the ability to seamlessly install and uninstall aspects in the running system. Secondly, the overhead of executing DIE advice consists of a minimal increment on the runtime overhead of PHANtom. Thirdly, as DIE is an embedded DSL standard Smalltalk tools can be used in the development and maintenance of plugins that use DIE.

### 5.6.1 Implementation Overview

A DIE pointcut expression is evaluated as a normal Smalltalk expression. Each domain-specific join point of Figure 2 is a Smalltalk class, subclass of DIEPointcut. DIEPointcut understands the messages result, parent and window. Sending these messages returns an instance that understands the messages isCode, =, & and |, again returning the appropriate object. As a result of this, evaluating a DIE pointcut returns a DIEPointcut subclass that is responsible for specifying its translation to a PHANtom pointcut as well as how its properties need to be extracted from the context (as specified in Figure 2).

For example the keyboard shortcut pointcut is defined in the ShortcutKey class, the relevant methods of which are given below. First, the primPhPointcut class specifies which is the equivalent PHANtom pointcut for a control key being pressed. The next three methods specify the code for extracting the parent, result and window parameter, respectively.

```
ShortcutKey>>primPhPointcut
    ↑PhPointcut
        receivers: 'KeyboardEvent'
        selectors: 'scanCode:'
        context: #(#receiver)
        if: [ :ctx | (ctx receiver isKeystroke) and:
            [(ctx receiver controlKeyPressed )or: [ctx receiver commandKeyPressed]]].

ShortcutKey class>>contextExtractorParent
    ↑[:ctx |  ctx receiver hand keyboardFocus.]

ShortcutKey class>>contextExtractorResult
    ↑[:ctx | ctx receiver keyString]

ShortcutKey class>>contextExtractorWindow
    ↑[:ctx |  SystemWindow topWindow]
```

These last three methods specify how to obtain the context parameters from a PHANTom runtime context object. For example, the contextExtractorResult method obtains the key being pressed by sending the keyString message to the KeyboardEvent instance that received the scanCode: message (indicated by ctx receiver). The context extraction methods are used as follows: when instantiating a DIE advice, these context extraction methods are used to obtain a one-argument block. Later, when the advice is actually run, these blocks will be executed with as argument a PHANtom runtime context object, yielding the respective context parameter.

DIEAdvice also is a class; it understands the before:do: and after:do: messages. Their first parameter is the pointcut object that has been obtained as described above, the second parameter is always a Smalltalk block. The implementation of these messages does the following:

1. Transform the pointcut to the equivalent PHANtom pointcut. This is performed by asking the DIE pointcut object for its transformation to a PHANtom pointcut.

2. Using Pharo's introspection abilities, the advice block is inspected for the names of its variables in order to know what information it needs.

3. This information is used to collect the required context extraction blocks, each as specified by the pointcut object and responsible for extracting one piece of context information at runtime and passing it to the advice block.

4. Finally, a PHANtom advice is built that has as the pointcut the result from the first step and as advice the result of the third step.

All of the above is performed once, when the DIE advice is instantiated. Only after the aspect to which the advice belongs is installed, the DIE advice effectively becomes active: a PHANtom aspect that contains the PHANtom advice created in the fourth step above is added to the system. In PHANtom, installation and uninstallation of aspects is performed dynamically on the running system, in a seamless fashion [Fabry and Galdames(2013)]. For an installed aspect, whenever its pointcut matches it will run its advice. DIE aspects will therefore run their before or after advice. For instance, for a DIE after advice this means executing the self-explanatory method given below:

```
DIEAdvice>>evalAfter: aContext
    |actuals retval|

    "build the list of actuals for the advice. Note that while this is an after advice
     it still happens *before* the original behavior is invoked"
    actuals := self argumentbuilders collect: [:builder | builder value: aContext].
    "invoke the original behavior"
    retval := aContext proceed.
    "call the block specified by the programmer, giving it the actuals we computed beforehand"
    self advice valueWithArguments: actuals.
    ↑retval
```

On the UI side, UIAdvice is a class that abstracts away the necessary knowledge to alter a window in order to add buttons to it. This includes finding the location where the buttons need to be added, the creation of actual buttons, and ensuring that changes to the window propagate properly to the buttons. As we mentioned above, further versions of DIE will expand and generalize UIAdvice and pointcuts to support more changes to windows.

### 5.6.2 Performance and Maintainability

The design assumption of DIE is that installation and uninstallation of DIE aspects is not a frequent activity, compared to the actual running of DIE advice. In order to minimise the overhead of DIE, hence as much processing as possible is performed at aspect instantiation and installation time. Costly reflective operations, such as extracting the names of parameters of the DIE advice block, are performed at aspect installation time. At runtime the only overhead, in addition to the overhead of PHANtom, is the `evalAfter:` method detailed above. Besides executing the original behavior captured by the advice, it builds the list of actual parameters, and runs the advice behavior specified by the programmer. It is therefore safe to say that except for overhead present is due to the PHANtom infrastructure, the overhead of DIE is kept to the minimum required to actually execute the programmed behavior.

The standard Smalltalk tools can be used in development and maintenance of plugins using DIE, thanks to the fact that DIE is an embedded DSL. Development tasks such as code browsing, writing unit tests and debugging are simply performed with the standard tools. Considering the overhead for code maintenance due to DIE being an aspect language, this overhead essentially reduces to a subset of the overhead of the use of aspect-oriented software. DIE pointcuts effectively encapsulate the complex implementation details of when to hook into the IDE's behavior, hence the developer is shielded from this source of complexity when using aspects. The only element that remains exposed to the developer is the behavior of DIE advice.

Debugging such DIE advice is just a task of similar complexity as debugging normal code. Consider, for example, the case of a bug in the behavior of DIE advice. The standard Smalltalk debugger is used, where an examination of the call stack will show that the `evalAfter:` method of `DIEAdvice` is present in the stack. This firstly serves as an indication that the code being debugged is a consequence of DIE advice executing. Secondly, it allows for navigation to the actual DIE advice that was executed, which allows it to be inspected, again allowing for further navigation to the DIE aspect that contains this advice, and so on.

## 6 Discussion

In this section, we take a step back and discuss higher-level characteristics of DIE. We first provide a comparison with publish/subscribe, second discuss the expressiveness of DIE, and third consider the generalizability of the approach to other approaches/IDEs.

### 6.1 DIE vs publish/subscribe

Publish/subscribe systems are the established alternative to implement plugins reacting to user actions. In these architectures, the plugin registers its interest to the set of events it wants to be informed about. The IDE will then notify the plugin of the relevant activity when an event matching the plugins' interest happens. This is usually done through a set of callback functions or methods of the plugin that the IDE executes.

### 6.1.1 Advantages over publish/subscribe systems

With respect to a publish/subscribe architecture, DIE possesses the following advantages:

**Modularity:** Using AOP techniques such as inter-type declarations, DIE allows modular extensions to existing classes when this provides a cleaner design. We use this in the "browsing history" example, as browsing previously visited methods is a behavior that inherently belongs to the code browser itself, since each browser has a separate history.

**Relating program execution points:** Recall that the "browsing history" example also relates various points in the execution of the program: When the buttons to show the next or previous method in the history are pressed, showing this method does *not* result in that method to be added to the history. This is performed in a straightforward fashion, using the standard AOP concept of `cflow`. This concept is absent in publish/subscribe.

**Extensibility:** DIE allows unforeseen extensions when needed. The publish/-subscribe model inherently relies on the correct set of events being defined. In case a necessary event is not exposed—or not exposed with enough precision—, the extension is not possible. In our examples, the plugin recording of visited methods needed to be aware of when it was active so as not to record spurious navigation. DIE allows a clean implementation of that unforeseen need.

**UI Extensions:** DIE allows the addition of new UI elements in existing windows, in the same conceptual layer (in response to IDE Events). Such a unification makes sense, since both notification and exposition of functionality to the user through UI elements are event-based. This reduces to the minimum necessary the number of concepts to handle, and is a factor in the conciseness of DIE's solution. We used this to expose the functionality of the history navigation plug-in through buttons, while other plugins used keyboard shortcuts (but could—similarly—use buttons).

**Conciseness:** As discussed above, DIE's syntax reduces boilerplate code, as it is tailored to its use case. An additional example is the fact that advice do not need to declare the parameters that they do not use: DIE automatically figures out which parameters are needed. In comparison, the event handling system present in Java relies on implementing anonymous subclasses of the event classes, and overriding the specific callback methods that the plugin needs, including their exact method signatures.

As an additional example of these advantages, we recall the focus events in the code completion example. The current version of OCompletion, Pharo's code completion tool, has a bug: a window losing focus is unable to notify the code completion menu that it should hide itself. As a workaround, the completion menu records its last user activity, and disappears after a long enough inactivity period. This is clearly far from ideal. In contrast, DIE allows this to be expressed concisely: the code reacting to focus events can easily access the UI and tell it to hide itself. Unlike the DIE implementation, the current implementation, the code completion UI is completely separate from the code browser itself, and can not do this in a concise or practical way.

A last example of the conciseness allowed by DIE is that, by allowing state and behavioral extensions to browsers in the history example, one does not need to maintain this state outside of the browser. This situation is more complex, as it adds additional data structures (maintaining a list of active browsers), and memory management issues (if a browser is closed, it needs to be removed from this lists, as it would not be garbage collected properly otherwise). The additional browser buttons would also need to reference this complex data structure to get a hold on the browser's history, additionally to the browser itself. By allowing this behavior to be part of the browser, DIE sidesteps these issues entirely.

#### 6.1.2 Drawbacks over publish/subscribe systems

Of course DIE does not always compare favorably with a more traditional publish/subscribe systems:

**New paradigm:** Publish/subscribe systems are well known, and as such are easier to use out of the box. On the contrary, DIE, as a Domain-Specific Aspect Language, requires users to familiarize with an array of novel concepts with an unfamiliar terminology. However, the choice of the different elements of the language was made so as to minimize this exposure.

**Extensibility:** The ability to extend tools through internal interfaces is a double-edged sword. The behavior that is relied upon may change from one version to the next. On the other hand, firstly the possibility of performing the extension in itself is valuable enough to warrant at least a cost/benefit analysis on a case-by-case basis. Secondly, such changes will typically only require a change to the pointcuts specified in the DIE program, the advice code will essentially remain the same.

### 6.2 The expressiveness of DIE

Is DIE expressive enough to cover a wide range of use cases? The anecdotal evidence we have gathered so far led us to think so. We presented four examples, using distinct functionality: Recent change notification; keyboard shortcuts; keyboard activity; focus activity; notification of opening windows of a certain type;

and extensions of existing tools, in state, behavior, and interface. This gives us confidence that the language has a large level of expressivity.

Further, the language itself is easily extensible. We took an organic approach when developping it, adding new concepts when experience proved them necessary. The process we used was to first implement missing functionality in terms of PHANtom aspects (DIE's fallback layer), and, if they were used frequently enough, to extend DIE to handle this case. For instance, our first version of the history example used a PHANtom pointcut to intercept the window construction and add buttons there; we made it more generic subsequently.

Most of the extensions we did in this manner involved defining new kinds of events to handle, *i.e.,*join points. Adding new join points in this way has shown to be an easy task. As such, we expect future changes, if they prove necessary, to be relatively straightforward.

The other area where probable extensions will be needed is on the UI side. So far, our UI advices are only capable of adding buttons to a button bar. There is nothing that theoretically can prevent such a scheme to be devised for other kinds of extensions; the only potential problems would be related to accidental complexity in the GUI framework Pharo uses, as we encountered when defining UI advice.

### 6.3 Generalizability of DIE

Another aspect of the discussion is how much DIE is generalizable to other programming languages, development environments, and user interfaces. We can not provide a definite answer to this, but give factors that could influence one way or another.

**Other Programming Languages:** DIE is implemented in Pharo Smalltalk, for the Pharo IDE. As such, it makes use of Smalltalk language features that are not necessarily available present in other languages. We specifically refer to the use of blocks and introspection on block parameters. Variants of DIE in languages that do not support theses are however still possible. For the former, the use of blocks could be replaced by some equivalent of function pointers, at the price of losing lexical scoping of variables in the blocks. We do not consider this as a real issue that lessens the applicability of DIE, note that none of the examples in this text rely on this scoping of variables. Regarding the latter; the use of introspection to determine the names of the variables in the advice block, this can be simply omitted. If such features are not available in the programming language, it suffices to extend the syntax of the DIE variant to require the programmer to specify a separate list of joint point properties that need to be passed to the advice.

**Other Aspect languages:** DIE is a layer on top of PHANtom, an aspect-oriented language for Pharo Smalltalk, using PHANtom as infrastructure for its core aspectual functionality. While PHANtom has advanced features (such as dynamic advice reordering), DIE does not make use of them. As such, a similar approach on top of another aspect language is possible. An optional non-standard requirement for this language, *i.e.,* beyond the standard pointcut-advice functionality of aspect languages, is to have dynamic deployment and undeployment of aspects. This allows plug-ins to be installed and uninstalled while the IDE is running. Without this feature, the IDE would need to be restarted whenever a plug-in is added or removed.

**Other IDEs:** The goal of DIE is to hide the accidental complexity present in event-handling systems in IDEs. As such, there is an implementation work to isolate the events of interest and capture them in an unified manner. Doing so is highly dependent on the actual implementation of the IDE.

**Other user interfaces:** Each IDE has a distinct user interface; some are wildly distinctive, such as in Code Bubbles [Bragdon et al.(2010a)] or as in Gaucho [Olivero et al.(2011)]. As such, exposing functionality in the user interface of each IDE will necessarily be an ad-hoc process that will be different in each case. However, one could argue that usability makes this treatment necessary anyways.

## 7 Related Work

To the best of our knowledge, there has been no work performed on extending IDEs with plugins in a domain-specific manner, *i.e.,* proposing a DSL for IDE extension. We are however aware of five DSALs that have been proposed for helping with the general development process: PCSL, Robust, Alert, PDL and CommentWeaver. We discuss them briefly next.

The Parameter Checking Specification Language; PCSL, allows for the cross-cutting concern of parameter specification checking to be specified [Bruntink et al.(2005)]. The goal in parameter checking is to validate parameters of functions before they are used, to obtain more reliable and robust systems. Examples of such validations are that input parameter should not be null, or that an output pointer parameter may not point to a location that already contains a value, in order to reduce the chance of memory leaks. The join point of the language is the passing of a parameter, pointcuts are function signatures and advice is written in C, where the thisParameter pseudovariable represents the parameter that is being passed.

A earlier langage for robust programming was unnamed, we term it Robust [Fradet and Südholt(1999)]. The Robust language provides a means for defining the standard and exceptional domain of variables. This is as robust programs should compute well-defined output values from well-defined sets of input values, and input values from the complement of this domain should cause errors to be raised. The language allows for the declaration of the handling of such exceptional situations in a modularized way. Join points are expressions in the program and assignments to variables, pointcuts are joined with advice in the definition of invariants, domain directives and overflow directives. The first define possible ranges for variables throughout the code, the second define possible values for variables at specific points in the code, the third define regions in the code with specific overflow or underflow handling.

With the Alert [Bagge et al.(2006)] language, the programmer can separately specify exception handling, both on the throwing side as on the catching side of the exception. The language is an embedded DSL that consists of the declaration of alert blocks in C. As such it may initially not be considered as a DSAL, however the authors state the following: One could consider our language extension as a domain-specific aspect language for error handling. [Bagge et al.(2006)] Join points in Alert are the throwing of an exception by a function call and the begin and the end of the execution of a method. Pointcuts are specified either by giving sets of functions or a single statement or block. Advice is the code of the exception handler, which may use extra keywords to retry the call that caused the exception or to provide a replacement value for that call.

In PDL [Morgan et al.(2007)], design rules can be expressed as an aspect program. Examples of design rules are stylistic guidelines, library usage rules, and correctness properties. Typically, design rule checkers take a design rule specification, and statically check a program for violations. PDL is a design rule checker where rules given are pointcut-advice pairs. Pointcuts express when a design rule is violated and advice states the corresponding error message. Joinpoints in PDL are static elements of the program, pointcuts use the typical syntax of AspectJ [Kiczales et al.(2001)], and advice consists of the error message that needs to be printed when the programmer is checked.

CommentWeaver [Horie and Chiba(2010)] allows for documentation of application programming interfaces (APIs) to be written in a modular way. The authors claim that much documentation contains cross-cutting concerns, resulting in documentation parts to be scattered thought the API documentation. This is addressed by Commentweaver providing special tags for modularly describing documentation elements. When the complete documentation is generated, the weaver appends these to the documentation of multiple methods, as specified by the tags. The join points are the static structure of methods, method calls and inheritance, although not on the program, but on the documentation. Pointcuts are the names of corresponding methods, or by AspectJ pointcut that identify methods statically. The advice specifies documentation text that needs to be added to other documentation parts.

It is clear that although all of these languages provide some domain-specific way to describe parts of the development process, none of them tackle the same issues as DIE: writing plug-ins for IDEs.

Outside of the above efforts, significant work in creating DSLs and DSALs that are linked to software development has been performed in the research area of DSLs. The effort in this area has been focused on providing support in the creation of DSLs. For example, the work of Rebernak et.al. referenced in the beginning of this section [Rebernak et al.(2009)] details the creation of two aspect languages for two different existing language development tools. The first aspect language allows for modular extension of the new language being created, while the second aspect language focusses on the creation of new tools for an existing language. This work, like other DSLs and DSALs in this area of research, is not focused on extending IDEs with plugins and hence is difficult to directly relate to DIE.

## 8 Conclusions and Future Work

Integrated Development Environments are frequently extended with new and unpredicted functionality. These new tools may have different information needs, and for this have to query the IDE in various ways. The number and heterogeneity of interfaces to query causes additional effort for the tool builders.

We presented a DSAL called DIE that aims to streamline this query work via the usage of a number of domain-specific join points. DIE allows one to define pointcuts on IDE event streams, as well as on the normal execution of the IDE. Advice are then defined to transmit the needed information to the tools and allow them to react accordingly. In order to display functionality to the user, pointcuts and advice can be used as well, either by implementing keyboard shortcuts, or altering the UI of the tools via buttons.

We have presented DIE by showing how it can be used to implement several IDE extensions: recent changes monitoring, code completion, window positioning, and browsing history. Finally, we provided a description of the domain-specific join point model for IDE events that is a part of DIE, provided the grammar of DIE, and described its implementation. We also extensively discussed how DIE compares to its main rival solution, publish/subscribe systems, and how expressive and general DIE is.

There are several ways in which DIE can be enhanced:

Our first avenue of future work is to generalize the concept of UI advice to permit richer modifications of existing tools, allowing for instance arbitrary widgets to be added, removed, or altered. We will also explore how to integrate our ideas with a presentation layer, for example Mondrian [Lienhard et al.(2007)] and Glamour [Bunge(2009)].

One characteristic we do not fully exploit yet is that events constitute streams. In some cases, patterns in the stream of events are more interesting than events themselves [Murphy et al.(2009)]. While these may be captured by the use of control flow constructs, as performed in Section 4.4, a higher-level abstraction would be more suitable. One such implementation for aspect-oriented programming is Tracematches by Allan *et al* [Allan et al.(2005)].

The system presented has been implemented as a prototype and could be further optimized. For example, when a DIE join point is captured by a pointcut, all context properties are filled in. However, this should only be performed for the properties that will be used in the advice. Fortunately this is a well-known problem in aspect-oriented language design, for which solutions have already been developed [Hilsdale and Hugunin(2004), Masuhara et al.(2003)].

To validate the language, we plan to realize more tools and extend the existing ones to be practically usable in real world development.

## Acknowledgments

This work was performed in the context of the INRIA Associated Team *PLOMO* (2013). Johan Fabry is partially funded by FONDECYT project number 1130253.

## References

[Allan et al.(2005)] Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: "Adding trace matching with free variables to AspectJ"; OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming systems and applications; 345–364; ACM, New York, NY, USA, 2005.

[Bagge et al.(2006)] Bagge, A. H., David, V., Haveraaen, M., Kalleberg, K. T.: "Stayin' alert:: moulding failure and exceptions to your needs"; Proceedings of the 5th international conference on Generative programming and component engineering; GPCE '06; 265–274; ACM, New York, NY, USA, 2006.

[Black et al.(2009] Black, A. P., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., Denker, M.: Pharo by Example; Square Bracket Associates, 2009.

[Bragdon et al.(2010a)] Bragdon, A., Reiss, S. P., Zeleznik, R. C., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., Jr., J. J. L.: "Code bubbles: rethinking the user interface paradigm of integrated development environments"; ICSE 2010: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering; 455–464; 2010a.

[Bragdon et al.(2010b)] Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., LaViola, J. J., Jr.: "Code bubbles: a working set-based interface for code understanding and maintenance"; CHI '10: Proceedings of the 28th international conference on Human factors in computing systems; 2503–2512; ACM, New York, NY, USA, 2010b.

[Bruntink et al.(2005)] Bruntink, M., van Deursen, A., Tourwe, T.: "Isolating idiomatic crosscutting concerns"; Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05; 37 – 46; 2005.

[Bunge(2009)] Bunge, P.: "Scripting browsers with Glamour"; (2009).

[Clayberg and Rubel(2008)] Clayberg, E., Rubel, D.: Eclipse Plug-ins (3rd Edition); Addison-Wesley Professional, 2008; 3 edition.

[Cleenewerck et al.(2008)] Cleenewerck, T., Noyé, J., Fabry, J., Lemeur, A.-F., Tanter, E.: "Summary of the third workshop on domain-specific aspect languages"; Proceedings of the 2008 AOSD workshop on Domain-specific aspect languages; DSAL '08; 1:1–1:5; ACM, New York, NY, USA, 2008.

[Douence et al.(2002)] Douence, R., Fradet, P., Südholt, M.: "A framework for the detection and resolution of aspect interactions"; GPCE'02: Proceedings of the 1st International Conference on Generative Programming and Component Engineering; 173–188; Springer-Verlag, 2002.

[Ducasse et al.(2006)] Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A. P.: "Traits: A mechanism for fine-grained reuse"; ACM Transactions on Programming Languages and Systems (TOPLAS); 28 (2006), 2, 331–388.

[Ekoko and Robillard(2008)] Ekoko, E. D., Robillard, M. P.: "Clonetracker: tool support for code clone management"; ICSE '08: Proceedings of the 30th international conference on Software engineering; 843–846; ACM, New York, NY, USA, 2008.

[Fabry and Galdames(2011)] Fabry, J., Galdames, D.: "PHANtom: a modern aspect language for Pharo Smalltalk"; Proceedings of the International Workshop on Smalltalk Technologies; IWST '11; 10:1–10:12; ACM Press, 2011.

[Fabry and Galdames(2013)] Fabry, J., Galdames, D.: "PHANtom: a modern aspect language for Pharo Smalltalk"; Software Practice and Experience; (2013); to Appear.

[Fowler and Parsons(2011)] Fowler, M., Parsons, R.: Domain-Specific Languages; Addison-Wesley, 2011.

[Fradet and Südholt(1999)] Fradet, P., Südholt, M.: "An aspect language for robust programming"; International Workshop on Aspect-Oriented Programming at ECOOP'99; 1999.

[Goldberg(1984)] Goldberg, A.: Smalltalk 80: the Interactive Programming Environment; Addison Wesley, Reading, Mass., 1984.

[Hilsdale and Hugunin(2004)] Hilsdale, E., Hugunin, J.: "Advice weaving in AspectJ"; AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development; 26–35; ACM, New York, NY, USA, 2004.

[Horie and Chiba(2010)] Horie, M., Chiba, S.: "Tool support for crosscutting concerns of api documentation"; Proceedings of the 9th International Conference on Aspect-Oriented Software Development; AOSD '10; 97–108; ACM, New York, NY, USA, 2010.

[Hudak(1996)] Hudak, P.: "Building domain-specific embedded languages"; ACM Computing Surveys; 28 (1996).

[Kersten and Murphy(2006)] Kersten, M., Murphy, G. C.: "Using task context to improve programmer productivity"; SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering; 1–11; ACM Press, New York, NY, USA, 2006.

[Kiczales et al.(2001)] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G.: "An overview of AspectJ"; Proceedings ECOOP 2001; number 2072 in LNCS; 327–353; Springer Verlag, 2001.

[Kiczales et al.(1997)] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: "Aspect-Oriented Programming"; M. Aksit, S. Matsuoka, eds., Proceedings ECOOP '97; volume 1241 of LNCS; 220–242; Springer-Verlag, Jyvaskyla, Finland, 1997.

[Kosar et al.(2012)] Kosar, T., Mernik, M., Carver, J. C.: "Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments"; Empirical Software Engineering; 17 (2012), 3, 276–304.

[Lienhard et al.(2007)] Lienhard, A., Kuhn, A., Greevy, O.: "Rapid prototyping of visualizations using mondrian"; Proceedings IEEE International Workshop on Visualizing Software for Understanding (Vissoft'07); 67–70; IEEE Computer Society, Los Alamitos, CA, USA, 2007.

[Masuhara et al.(2003)] Masuhara, H., Kiczales, G., Dutchyn, C.: "A compilation and optimization model for aspect-oriented programs"; G. Hedin, ed., Compiler Construction; volume 2622 of Lecture Notes in Computer Science; 46–60; Springer Berlin / Heidelberg, 2003.

[Mernik et al.(2005)] Mernik, M., Heering, J., Sloane, A. M.: "When and how to develop domain-specific languages"; ACM Comput. Surv.; 37 (2005), 4, 316–344.

[Morgan et al.(2007)] Morgan, C., De Volder, K., Wohlstadter, E.: "A static aspect language for checking design rules"; Proceedings of the 6th international conference on Aspect-oriented software development; AOSD '07; 63–72; ACM, New York, NY, USA, 2007.

[Murphy et al.(2006)] Murphy, G. C., Kersten, M., Findlater, L.: "How are Java software developers using the Eclipse IDE?"; IEEE Software; (2006).

[Murphy et al.(2009)] Murphy, G. C., Viriyakattiyaporn, P., Shepherd, D.: "Using activity traces to characterize programming behaviour beyond the lab"; Proceedings of ICPC 2009 (17th IEEE International Conference on Program Comprehension; 90–94; 2009.

[Olivero et al.(2011)] Olivero, F., Lanza, M., D'Ambros, M., Robbes, R.: "Enabling program comprehension through a visual object-focused development environment"; G. Costagliola, A. J. Ko, A. Cypher, J. Nichols, C. Scaffidi, C. Kelleher, B. A. Myers, eds., VL/HCC 2011: Proceedings of the 27th IEEE Symposium on Visual Languages and Human-Centric Computing; 127–134; IEEE, 2011.

[Rebernak et al.(2009)] Rebernak, D., Mernik, M., Wu, H., Gray, J.: "Domain-specific aspect languages for modularising crosscutting concerns in grammars"; IET Software; 3 (2009), 3, 184–200.

[Robbes and Lanza(2010)] Robbes, R., Lanza, M.: "Improving code completion with program history"; Journal of Automated Software Engineering; 17 (2010), 2, 181–212.

[Robbes et al.(2010)] Robbes, R., Pollet, D., Lanza, M.: "Replaying ide interactions to evaluate and improve change prediction approaches"; Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories); 161 – 170; IEEE CS Press, 2010.

[van Deursen et al.(2000)] van Deursen, A., Klint, P., Visser, J.: "Domain-specific languages: An annotated bibliography"; SIGPLAN Notices; 35 (2000), 6, 26–36.