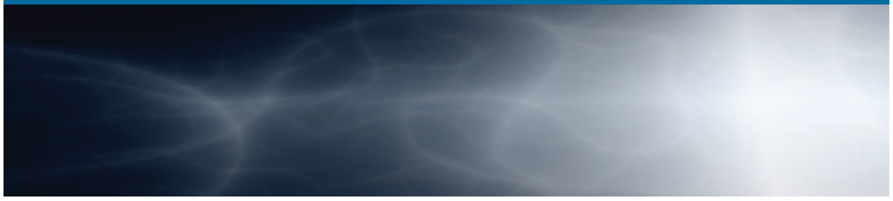
FEATURE: **PROGRAMMING TOOLS**

Programming with Ghosts

Oscar Callaú and Éric Tanter, University of Chile

// To support incremental programming, integrated development environments should support *ghosts*: reifications of undefined entities built automatically and nonintrusively, based on their usage. //



TEST-DRIVEN DEVELOPMENT HAS promoted the practice of writing test cases first, before implementing the actual system that will fulfill these tests.[1] New features are specified by their corresponding test cases. Programming then consists of defining and completing the system such that all the tests pass. Similarly, in traditional top-down programming, you write a procedure to address a given problem by relying on smaller auxiliary procedures that might not yet be implemented.

In both top-down programming and test-driven development, programming is incremental. Programmers write code that uses entities that are either not yet defined or only partially defined. Considering that these approaches are part of software development's daily practice—a Web survey of nearly 300 practitioners reports that more than half use test-driven development[2] and that agile and iterative approaches to software development have become very popular[3]—you might expect modern integrated development environments (IDEs) to properly support this programming style.

However, modern IDEs are mostly unable to support a pure incremental programming style. If a program uses an entity that's undefined—a procedure, method, class, interface, and so on—the only feedback the IDE provides is an error. Sure enough, you usually get the opportunity for the IDE to automatically create a matching code skeleton, but this feature—as we will see—is tedious, obtrusive, and limited. Several recent IDE enhancements and third-party plug-ins are devoted to better supporting incremental programming, further hinting at the practical relevance of the problem. However, we will show that even these state-of-the-art tools don't provide a seamless experience.

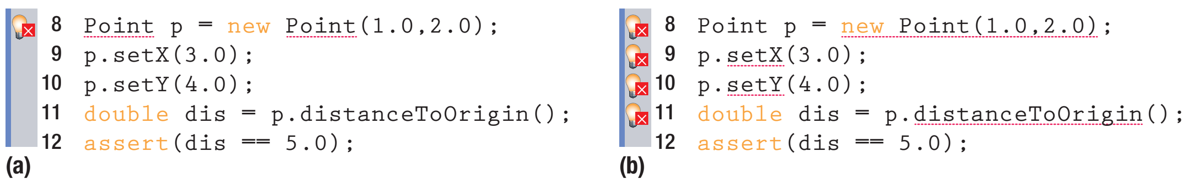
We propose a very simple idea to address this issue. Instead of only reporting error messages, we propose that IDEs transparently and nonintrusively build a reification of undefined entities according to their usage, progressively refined as the program is elaborated. The developer can then concentrate on the task at hand while reasoning about undefined entities through useful feedback on the constraints and dependencies that their usage implies. Once the programmer is ready to implement these entities, the IDE can provide an appropriate code skeleton that matches all the inferred dependencies. We call these reified, undefined entities *ghosts*, and we argue that IDEs should support them to assist developers in incremental development.

The State of the Practice in IDEs

To fully grasp the problem, let's first look closely at the state of the practice with modern IDEs. We first focus on Eclipse because it's arguably the most widely used Java IDE,[4] and it illustrates the issues related to incremental programming.

Errors, Errors, Errors ...

Suppose we have to provide a class `Point` such that point objects are located on a plane and their distance to the origin is computable. We can start by writing a first test case, as in Figure 1a.



```
 8  Point p = new Point(1.0,2.0);
 9  p.setX(3.0);
10  p.setY(4.0);
11  double dis = p.distanceToOrigin();
12  assert(dis == 5.0);
```
(a)

```
 8  Point p = new Point(1.0,2.0);
 9  p.setX(3.0);
10  p.setY(4.0);
11  double dis = p.distanceToOrigin();
12  assert(dis == 5.0);
```
(b)

FIGURE 1. Defining a test case for undefined class **Point**: (a) The icon on the margin of line 8 flags two errors and links to a suggestion for fixing them; (b) the same test case after generating a skeleton of class **Point** now includes three more lines with errors.

Because class **Point** isn't yet defined, the IDE marks two errors on line 8. In the margin, the IDE provides an icon that links to a suggestion for fixing the error; in this case, the IDE offers to create class **Point**. If the developer follows this suggestion, the IDE reports yet more errors (Figure 1b). These errors, on lines 8–11, report undefined constructors and members of class **Point**. Here again, the IDE suggests generating code skeletons. However, these suggestions must be accepted one by one. The IDE doesn't "understand" from the test-case definition that we want to define a single class with a constructor and three methods.

Options for Handling Errors

Attending to each and every error message related to an undefined entity breaks the flow of programming. It requires the programmer to explicitly trigger and validate the suggestion of generating a code skeleton. Additionally, it invariably triggers a context switch in the editor, bringing the file in which the skeleton is generated to the forefront, and so taking the programmer away from the original code she was writing. Even worse, if the programmer elects to generate a class or interface, it not only triggers an editor-context switch but also first shows a wizard window.

To illustrate, this implies the following steps in the test case of class **Point**:

- from the test-case buffer, request the generation of class **Point**;
- fill in the creation wizard;
- end up in the buffer of class **Point**;
- navigate back to the test-case buffer;
- request the generation of the constructor;
- end up in the buffer of class **Point**;
- navigate back to the test-case buffer;
- request generation of method **setX**;
- end up in the buffer of class **Point**;
- navigate back to the test-case buffer; and
- repeat the process similarly for the **setY** and **distanceToOrigin** methods.

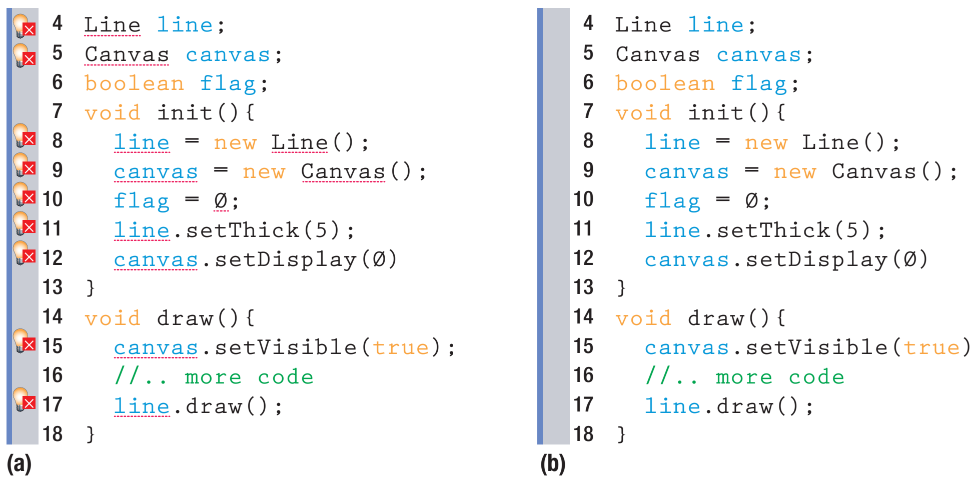
The cognitive burden of these context switches often invites programmers to the alternative of ignoring all error messages until they reach a stable point in their implementation, when they can address them all in a batch. IDEs even include an option to deactivate the error markers altogether so that programmers can effectively concentrate on the main task before bothering with the definition of auxiliary classes and methods.

The problem with both these options is that actual type errors go unnoticed, defeating the purpose of type feedback in the IDE. Consider, for instance, the code of Figure 2. There are many errors reported in this code, due to the use of undefined classes **Line** and **Canvas**. Turning off error markers (either by deactivating them in the IDE or by not paying attention to them) means that the actual type error on line 10 remains unnoticed.

Apprehending Undefined Entities

Figure 2 also reveals another limitation of current IDEs. There are two undefined classes used in this example and 11 marked errors, yet only one is an actual type error. Of the 10 remaining errors, some are related to **Line** and the others are related to **Canvas**. Because all errors are marked similarly, it's hard to correlate them appropriately. Moreover, errors related to a single undefined entity, such as **Line**, are scattered across several methods.

Similarly, within a given method, all error markers are tangled together: **init** has markers related to **Line**, to **Canvas**, and to the actual type error. Scattering and tangling make it hard to apprehend the dependencies on an undefined entity such as **Line**. Typical tasks such as reasoning about an entity's interface are impossible while that entity is undefined.



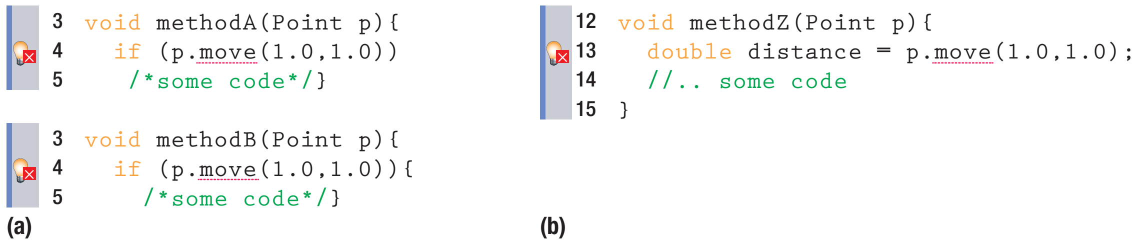
(a)

```
 4 Line line;
 5 Canvas canvas;
 6 boolean flag;
 7 void init(){
 8   line = new Line();
 9   canvas = new Canvas();
10   flag = 0;
11   line.setThick(5);
12   canvas.setDisplay(0)
13 }
14 void draw(){
15   canvas.setVisible(true);
16   //.. more code
17   line.draw();
18 }
```

(b)

```
 4 Line line;
 5 Canvas canvas;
 6 boolean flag;
 7 void init(){
 8   line = new Line();
 9   canvas = new Canvas();
10   flag = 0;
11   line.setThick(5);
12   canvas.setDisplay(0)
13 }
14 void draw(){
15   canvas.setVisible(true);
16   //.. more code
17   line.draw();
18 }
```

FIGURE 2. IDE error reporting options. Actual type errors, such as the one on line 10, are hard to distinguish whether the error markers are (a) activated or (b) deactivated.



(a)

```
void methodA(Point p){
  if (p.move(1.0,1.0))
   /*some code*/}
```

```
void methodB(Point p){
  if (p.move(1.0,1.0)){
    /*some code*/}
```

(b)

```
void methodZ(Point p){
  double distance = p.move(1.0,1.0);
  //.. some code
}
```

FIGURE 3. Inconsistent type usages: (a) two consistent usages of method **move** and (b) an inconsistent usage. The error markers only refer to the fact that the method isn't defined.

What about Consistency?

Considering undefined entities only as isolated errors brings some more problems in practice: IDEs don't detect type inconsistencies until they generate skeletons. Consider the code in Figure 3a. The undefined method **move** on class **Point** is twice used consistently: in both cases, arguments are of type **double** and the return type is **boolean**.

Now consider the extension shown in Figure 3b. Method **move** is used again, but this time it's expected to return a **double**. The IDE can only detect that **move** isn't defined and offer the possibility to generate it. It can't report that these three usages aren't consistent with each other. If the programmer chooses to generate **move** using the marker of **m3**, then **move** is generated with return type **double**. Of course, this causes two error messages for the other methods, indicating that **double** can't be converted to **boolean**. The IDE can't relate errors associated to undefined entities.

IDE Support for Incremental Programming

We studied several other major IDEs besides Eclipse for Java to determine how well they support incremental programming: Microsoft's Visual Studio 2010 (C#), IntelliJ's IDEA 11 (Java), and Oracle's NetBeans 7.0.1 (Java). We also looked at some popular IDE plug-ins—namely ReSharper (www.jetbrains.com/resharper) and CodeRush,[5] both for Visual Studio—because of their interesting features.

Considering the programming workflow with undefined entities, we formulated several questions related to the definition, verification, and generation of undefined entities. Table 1 summarizes our findings. For each question, we give the default answer and highlight the notable exceptions. More details, including specific code snippets and snapshots for each IDE and plug-in, can be found online.[6]

TABLE 1

How major IDEs support programming with undefined entities.

Questions	Default answer	Notable features
Definition		
How are undefined types reported in the IDE?	As errors	
Can undefined types be distinguished easily?	No	IDEA, ReSharper, and CodeRush report all undefined entities in a specific manner.
Are members of undefined types also reported?	No	IDEA, ReSharper, and CodeRush do report them.
Is it possible to easily identify the current set of undefined members associated to a single (undefined) type?	No, all undefined entities are signaled similarly	
How are undefined members of external libraries reported?	Like any undefined members	CodeRush reports them as actual errors, similar to type errors.
Verification		
Is the use of undefined entities subject to type checking?	No	IDEA and ReSharper only check some argument types and some local variable assignments
Generation		
Can an undefined entity be fully generated in a single click?	No	NetBeans, IDEA, VisualStudio, and ReSharper can only generate a class with a single constructor at once; Visual Studio can generate fields for constructor arguments; CodeRush can generate a type with several members at once, but only based on the usages in the current file.
Does generation force a switch to the buffer of the new entity?	Yes	VisualStudio can generate in the background.

This study shows that programming with undefined entities is currently not well supported. Even new features, such as Visual Studio's Generate From Usage[7,8] or plug-ins like CodeRush and its support for Consume-First Development,[5] fall short of addressing the whole picture, even though they explicitly target incremental programming.

Programming with Ghosts

We propose to make undefined entities explicit in the IDE, calling them ghosts and exploiting them through the IDE metaphors that programmers are used to. This simple idea turns out to address all the issues raised earlier. Ghosts are nonintrusive. They can be defined on the fly, used to reason about partially defined code, checked for type consistency early, and used to generate full code skeletons when needed.

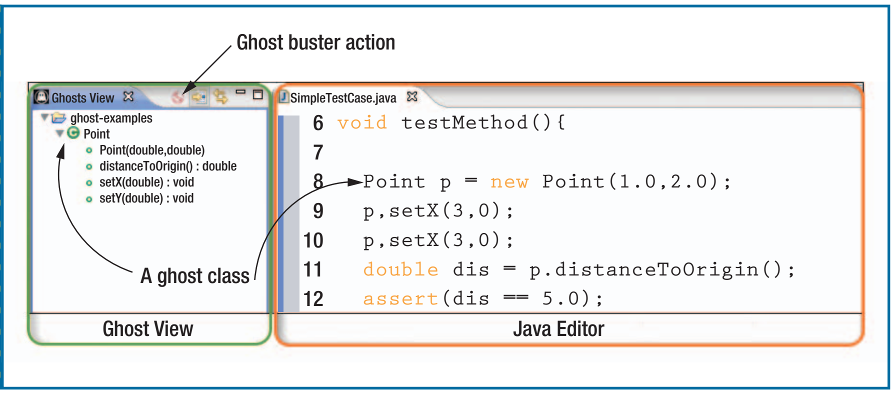
Ghosts: Reifying Undefined Entities

Instead of letting undefined entities manifest in the IDE in the form of errors, we reify them as ghosts. A ghost class is a class that's used but not yet defined. Similarly, the IDE can create ghost interfaces, ghost methods, ghost constructors, and ghost fields.

The IDE can display ghosts just as it does defined entities. Figure 4 shows the Ghost View of the Eclipse plug-in we developed: class `Point` is being used with a two-argument constructor and methods `setX`, `setY`, and `distanceToOrigin`. Because ghosts appear in their own separate view, the IDE doesn't have to report their usages as errors. This approach avoids visual noise so that actual type errors clearly manifest.

Building Ghosts

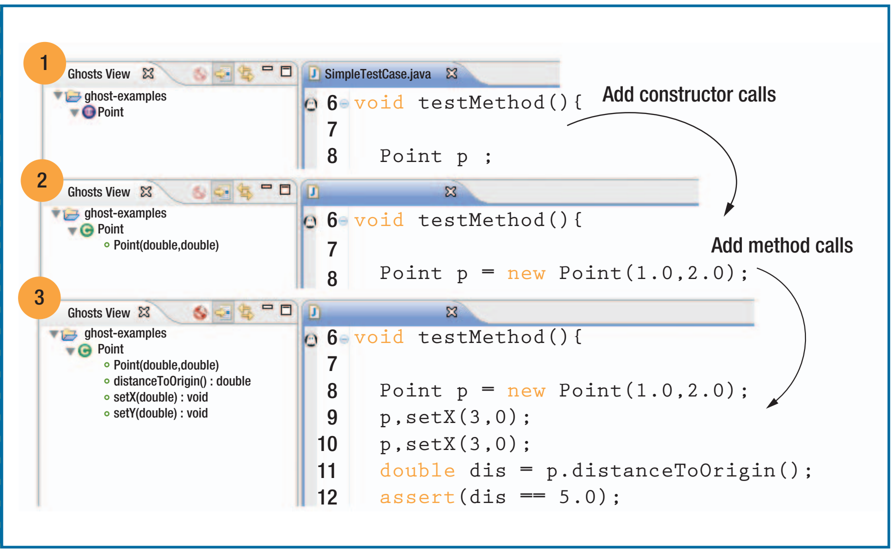
Programmers create and refine ghosts on the fly, as they write a program. There's no need for explicit (and disruptive) user actions. Figure 5 shows how the Ghost View evolves with the code. At first, only `Point` appears in the view, as an interface. The `testMethod` is marked to indicate that it relies on a ghost. Once the programmer writes code that instantiates `Point`, the view updates. It's clear that `Point` should be a class, not an interface, and that it ought to have a matching constructor. As the programmer writes the test case, the



```
void testMethod(){

   Point p = new Point(1.0,2.0);
   p,setX(3,0);
   p,setX(3,0);
   double dis = p.distanceToOrigin();
   assert(dis == 5.0);
```

FIGURE 4. Ghost View in Eclipse. Undefined entities used in the Java Editor appear in the Ghost View. The user can select a ghost in the Ghost View and generate a full code skeleton by selecting the Ghost buster action.



```
void testMethod(){

   Point p ;
```

```
void testMethod(){

   Point p = new Point(1.0,2.0);
```

```
void testMethod(){

   Point p = new Point(1.0,2.0);
   p,setX(3,0);
   p,setX(3,0);
   double dis = p.distanceToOrigin();
   assert(dis == 5.0);
```

FIGURE 5. Creating and refining ghosts on the fly. First, the IDE infers **Point** as a ghost interface; because **Point** is instantiated, it's replaced by a ghost class with the corresponding constructor signature; finally, the IDE adds methods as soon as they're used.

IDE adds the methods used on **Point** to the ghost, reflecting the programmer's intent in real time.

Creating ghosts implies performing some analysis of the code as it's written. For instance, when a developer introduces an undefined type, the IDE assumes that it corresponds to an interface. (It's well-recognized that programming against interfaces should be favored over classes.[9]) But this decision should be reverted if it turns out that only a class makes sense—for instance, if the entity is instantiated.

Similarly, defining ghost members requires performing some local type inference[10] to deduce member signatures from their usage context. Our implementation relies on an inference process inspired by the algorithm of Weiyu Miao and Jeremy Siek.[11] The algorithm type-checks code fragments on the fly, creating constraints for each occurrence of a ghost entity. At usage sites, the algorithm unifies the set of constraints associated to a ghost entity to infer the most general type, if any.

Checking Ghosts

Inferring ghost member types according to their usage context enables early detection of type inconsistencies. For instance, consider again the example of Figure 3, in which the **move** method is used in two incompatible ways: once with a **boolean** return type and once with a **double** return type. The Ghost View shows two methods—each with a different return type—and reports an error because Java doesn't allow methods of the same name to have different return types (see Figure 6).

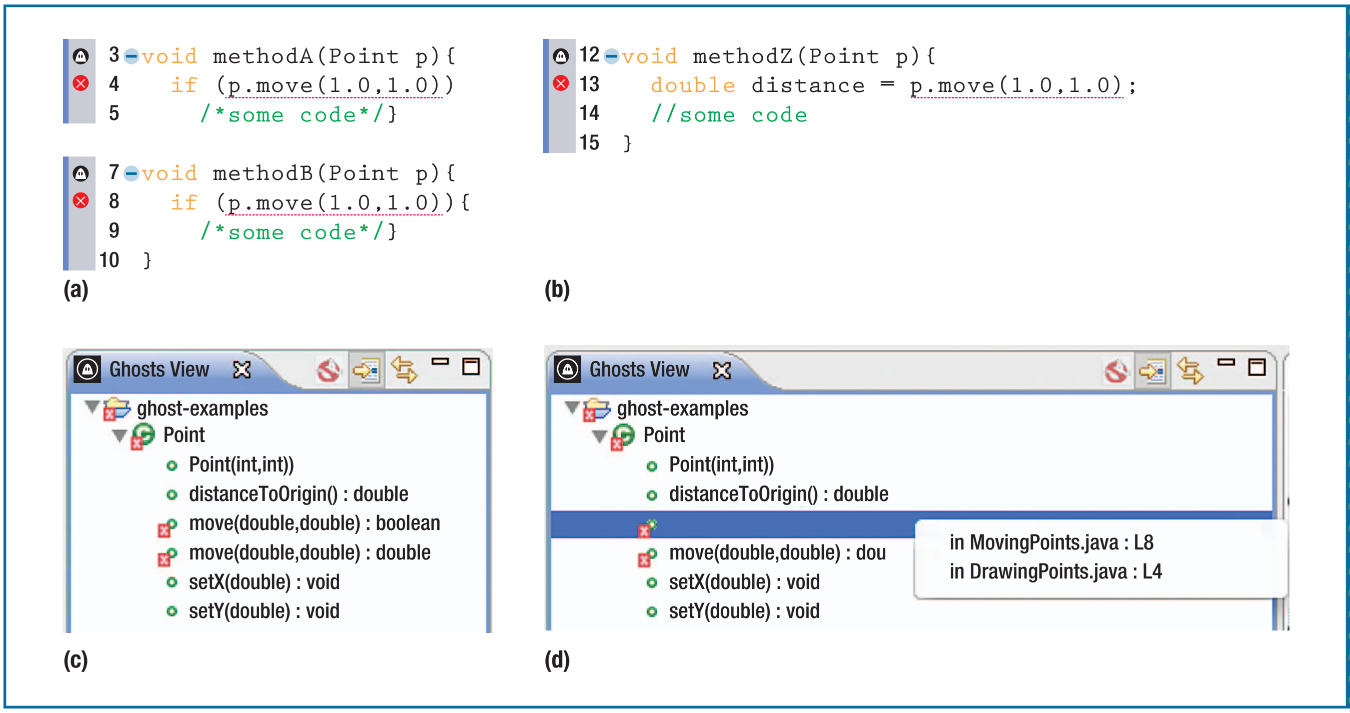
The programmer can exploit type feedback flexibly. For instance, if the programmer uses the **setX** method sometimes with a **double** argument and sometimes with an **int** argument, two ghost methods appear in the Ghost View. The programmer can leave both methods—it's perfectly valid to have such overloaded methods—or correct some uses to have a single ghost method.

Connecting Ghosts to Source Code

Navigating from a specific ghost to all the code locations where it's used makes it easy to correct erroneous code that creates an undesired ghost. For instance, in the last example, programmers that want to fix the usages of **move** to comply with return type **double** can identify all the code locations that imply the undesired ghost method (see Figure 6c).

Busting Ghosts

After the programmer has completed the main task and is happy with the corresponding ghosts, she can selectively "bust" them into actual definitions using the ghostbuster button (see Figure 4). This mechanism relies on the standard capabilities of the IDE to generate skeletons. The difference is that ghosts per-


```
void methodA(Point p){
   if (p.move(1.0,1.0))
      /*some code*/}

void methodB(Point p){
   if (p.move(1.0,1.0)){
      /*some code*/}
}
```
(a)

```
void methodZ(Point p){
   double distance = p.move(1.0,1.0);
   //some code
}
```
(b)

Ghosts View
ghost-examples
Point
Point(int,int))
distanceToOrigin() : double
move(double,double) : boolean
move(double,double) : double
setX(double) : void
setY(double) : void

(c)

Ghosts View
ghost-examples
Point
Point(int,int))
distanceToOrigin() : double
move(double,double) : dou
setX(double) : void
setY(double) : void
in MovingPoints.java : L8
in DrawingPoints.java : L4

(d)

FIGURE 6. Reporting type errors. The IDE reports type errors (a) in code editors and (b) in the Ghost View, which supports (c) navigation to the source location. There is no need to request generation of ghosts in order to get type feedback about them.

mit the generation of the whole skeleton for a given class (including all its members) or even all the skeletons for all the current project's ghosts. Ghost generation happens in the background, without switching context. Busted ghosts simply disappear from the Ghost View and appear as normal entities in the project explorer view.

Undesired Ghosts

Sometimes, programmers accidentally use undefined entities. Our approach is conceptually limited by default, because these unintentional uses are reflected as ghosts, which then must be eliminated. For instance, a spelling mistake like `movee` instead of `move` manifests as a ghost method; if it's undesired, the programmer can navigate from the ghost to its occurrence in the code and fix it. Another example is missing import statements. Using `List` without importing `java.util.List` results in a ghost interface; right-clicking the ghost offers the possibility of importing the interface from the library instead.

To limit the number of undesired ghosts, our tool doesn't create ghost members on classes that are external to the current project. For instance, calling `leength` on a string object yields an error, not a new ghost method on `String`. For imports, our tool also supports a user-specified list of names that should never give rise to ghosts, such as `List` or `Map`, hence reverting the default behavior selectively.

Despite being a simple idea, ghosts offer significant contributions to current IDE capabilities. The Ghost plug-in for Eclipse and the Ghost extension for Smalltalk Pharo—showing the potential of ghosts for dynamic languages—are available online, together with supplementary material.[6] The Eclipse plug-in is a proof of concept that doesn't yet support several Java features, most notably generics and exceptions. The tool also does not currently try to infer super types of ghosts or reasoning about method-call chains. These limitations ought to be explored and addressed in production-quality ghost implementations but don't represent fundamental restrictions of the ghost concept.

Ghosts are a simple and useful metaphor to better support incremental development. Their potential is yet to be fully exploited. In particular, IDE features such as code completion and refactoring can make productive use of ghosts.

References

1. K. Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2002.
2. S.W. Ambler, "How Agile Are You? 2010 Survey Results," Scott W Ambler & Assoc.,

ABOUT THE AUTHORS

OSCAR CALLAÚ is a PhD student at the University of Chile in the Pleiad research laboratory. His research interests include type systems, integrated development environments, and mining software repositories. Callaú received BSc degrees in both computer science and computer engineering from Major University of San Simeon at Cochabamba, Bolivia. He's a member of the ACM. Contact him at oalvarez@dcc.uchile.cl.

ÉRIC TANTER is an associate professor in the computer science Department of the University of Chile, where he co-leads the Pleiad research laboratory. His research interests include programming languages and tool support for modular and adaptable software. Tanter received a PhD in computer science from both the University of Nantes and the University of Chile. He's a member of IEEE and the ACM. Contact him at etanter@dcc.uchile.cl.

2010; www.ambysoft.com/surveys/howAgileAreYou2010.html.
3. C. Larman, *Agile and Iterative Development: A Manager's Guide*, Addison-Wesley Professional, 2003.
4. G. Goth, "Beware the March of this IDE: Eclipse Is Overshadowing Other Tool Technologies," *IEEE Software*, vol. 22, no. 4, 2005, pp. 108–111.
5. DevExpress, "CodeRush—Consume-First Development," tech. note, 2012; http://devexpress.com/Products/Visual_Studio_Add-in/Coding_Assistance/consume_first_development.xml.
6. O. Callaú and É. Tanter, "Ghosts," project website, 2012; http://pleiad.cl/ghosts.
7. Microsoft Developer Network, "Generate from Usage," tech. note, 2012; http://msdn.microsoft.com/en-us/library/dd409796.aspx.
8. Microsoft Developer Network, "Walkthrough: Test-First Support with the Generate From Usage Feature," tech. note, 2012; http://msdn.microsoft.com/en-us/library/dd998313.aspx.
9. J. Bloch, *Effective Java*, 2nd ed., Addison-Wesley, 2008.
10. B.C. Pierce, *Types and Programming Languages*, MIT Press, 2002.
11. W. Miao and J. Siek, "Incremental Type-Checking for Type-Reflective Metaprograms," *Proc. 9th ACM SIGPLAN Int'l Conf. Generative Programming and Component Eng.* (GPCE 2010), ACM, 2010, pp. 167–176.