# Supporting Incremental Programming with Ghosts

Oscar Callaú
PLEIAD Lab Computer Science Department (DCC) University of Chile
*oalvarez@dcc.uchile.cl*

***Abstract*—Best practices in programming typically imply coding using classes and interfaces that are not (fully) defined yet. However, integrated development environments (IDEs) do not support such incremental programming seamlessly. Instead, they get in the way by reporting ineffective error messages. Ignoring these messages altogether prevents the programmer from getting useful feedback regarding actual inconsistencies and type errors. But attending to these error messages repeatedly breaks the programming workflow.**

**In order to smoothly support incremental programming, we propose to extend IDEs with support of undefined entities, called *Ghosts*. Ghosts are implicitly reified in the IDE through their usages. Programmers can explicitly identify ghosts, get appropriate type feedback, interact with them, and *bust* them when ready, yielding actual code.**

## I. Introduction

Best practices in programming typically imply coding using modules, such as classes or functions that are not (fully) defined yet. Two typical examples of these practices are test-driven development [1] and top-down programming. In those two, programmers use entities that are either not yet defined or partially defined, *i.e.* programming is incremental.

Although incremental programming style is a daily practice of software development [2], modern IDEs do not properly support such a programming style. By-default most IDEs reference undefined entities as common errors, such as missing modules, however those IDEs offer the possibility to generate a code skeleton. Nevertheless, this feature is very limited and requires full programmer attention. Currently, there are several state-of-the-art tools (IDE enhancements and Plugins) [3]–[6] that are focused on solving this problem, however their solutions are partial and usually focus on a particular issue and not the whole problem.

To properly address this problem, we propose to extend the IDE in order to automatically and progressively build a reification of undefined entities, based on their usage. We call these reified entities *Ghosts* [7], [8]. With Ghosts, the IDE can provide a structured and condensed view of these entities with useful instant feedback, *e.g.* exposing possible inconsistent usages, based on their interactions. Then, the programmer can concentrate on one task at a time, without ending up on attending to disruptive messages. Furthermore, once the programmer finishes her main task, she can generate a whole code skeleton for one or all ghosts that satisfy all their dependencies.

## II. The State of the Practice

To better understand the problem and as a point of reference, we use the Eclipse IDE to illustrate average IDE approaches on supporting incremental programming. Later on, we expose outstanding solutions presented in some other popular IDEs and Plugins currently available.

Figure 1 (a) shows a starting point of coding for a programmer: the declaration and instantiation of two objects from undefined classes Person and Address. The IDE shows these instructions as errors, and as a solution, offers the possibility to generate an empty class skeleton for each missing class. However the programmer must manually trigger them, one by one. Even worse, each generation implies a context switch to the newly created class. Once the programmer generates these classes and continues coding, she soon will end up with more and more errors, see Figure 1 (b). The only feedback from the IDE is that those errors represent missing entities: constructors and methods. The IDE is unable to understand that the programmer is writing a series of methods and constructors that belong to classes Person and Address. Even worse, those errors are tangled and scattered, making it difficult to see real errors, such as in L26, or to distinguish between members of class Person and class Address. Furthermore, the IDE cannot detect inconsistencies between missing members until they are generated, *e.g.* inconsistent return type of method getPostalCode in L31 and L32. The only offered solution to this problem by the IDE is to generate, one by one, all missing entities to start to get useful feedback. As demonstrated in this example, the programmer ends up attending to a series of disrupted errors, one by one, resulting in a constant context switches.

Instead of attending to error messages, some programmers ignore all of them until they reach a stable point. Nevertheless, this approach does not solve any of above problems. In fact, both strategies, attending to errors and ignoring them, are not real solutions. In both cases:

1) Undefined entities are reported as errors.
2) Undefined entities from different classes cannot be distinguished easily.
3) Real errors, *e.g.* L26, go unnoticed.
4) The IDE cannot detect inconsistencies between undefined entities until they are generated, *e.g.* L31 & L32.
5) Undefined entities need to be generated one by one.
6) Each generation action implies a context switch.

Along with Eclipse 3.7.1, we analyze three major IDEs: Microsoft Visual Studio 2010, IntelliJ IDEA 11, and Oracle NetBeans 7.0.1. We also include two Plugins for Visual Studio: ReSharper 6.1 and CodeRush 12. The list of issues perviously reported can partially apply to these IDEs and

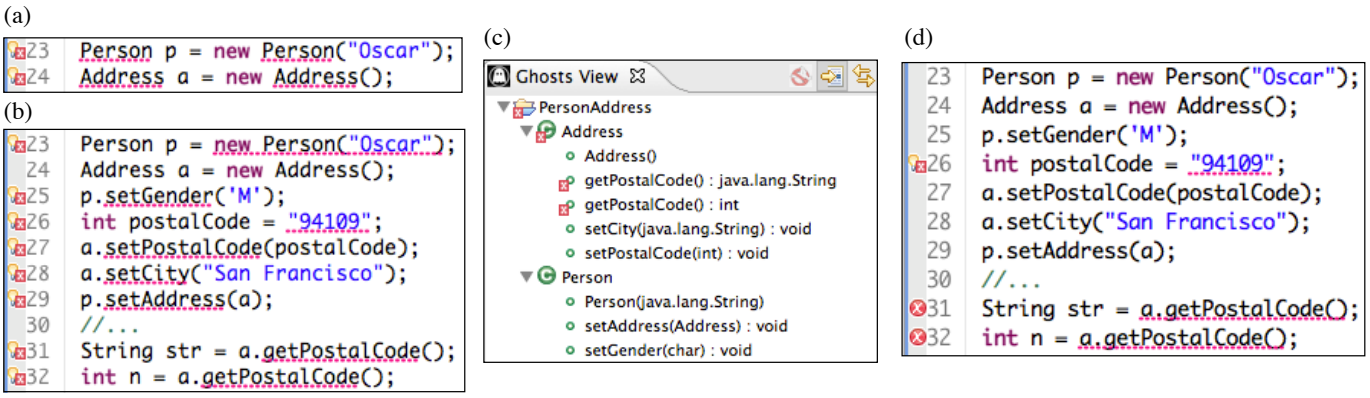


Fig. 1. **(a)** Programmer starts writing two instances of undefined classes Person and Address on the IDE. **(b)** Once programmers attend to initial errors and continue writing code on the IDE. **(c)** Ghost view and instant inconsistency feedback. **(d)** Same code as (b) after Ghosts clean up and check.

Plugins, however there are remarkable exceptions:

- IDEA, ReSharper and CodeRush report undefined entities in a specific manner.
- IDEA and ReSharper perform partial and simple type checking.
- NetBeans, IDEA, Visual Studio and ReSharper can generate a class with a single constructor. Visual Studio can include generation of fields based on constructor arguments. CodeRush can generate members, but only if they appear in the current file.
- Visual Studio can generate a code skeleton in the background.

A detailed comparison between these IDEs and Plugins, including snapshots and code snippets, can be found online [8].

## III. Programming with Ghosts

We extend Eclipse to automatically and progressively report on undefined entities usages, we call this extension *Ghosts*. This is accomplished by making undefined entities (ghosts) explicit in the IDE. Therefore, programmers can see them in a dedicated view. Furthermore, programmers can get useful feedback through type checking ghosts usages. In addition, programmers can interact with ghosts, *e.g.* navigating to all ghost usages across the current projects. And finally, programmers can generate ghosts (including members) in the background with a single step.

Ghosts can be inferred and refined on the fly from their usages. They can be interfaces (as default), classes, constructors, methods and fields. Once a ghost is reified, it is displayed as a valid entity in the *Ghost View*, a specific and condensed view for ghosts, see Figure 1 (c). From the point of view of the IDE, ghosts are not errors any more, therefore all error references are removed, avoiding visual noise. Creating and refining ghosts requires some analysis, *e.g.* ghost method signatures are inferred through a local type inference algorithm inspired by Miao and Siek [9]. Ghosts permits the generation (in the background) of a given ghost class with its members or even all ghosts related to the current project.

In some cases, the automatic reification of a ghost could not be useful for the programmer, *e.g.* a typo. Ghosts reduces this natural disadvantage by avoiding reifying ghosts methods from external projects or standard libraries; allowing programmers to import a class from an external library that matches a ghost class; and permitting programmers to define an explicit list of non-ghosts.

Figure 1 (d) shows how our Ghosts Plugin for Eclipse interact with the code defined in Section II, before the generation of any code. In addition, the Ghosts view, see Figure 1 (c), condenses all usage of ghosts classes and members in the current project. All non-useful error markers are removed highlighting real errors, such as L26. Ghost usages are type checked revealing a type inconsistency in the return type of method getPostalCode in L31 and L32. This inconsistency is clearly marked on the Editor and the Ghost view. Therefore, the listed issues in Section II are solved by the introduction of Ghosts.

**Related Concepts.** Mock Objects [10] are similar to Ghosts in the sense that mock objects are declarations of future object interfaces. However, the programmer explicitly provides mock objects, instead to an implicit reification in Ghosts. Prorogued Programming [11] is a new programming paradigm that, similar to ghosts, allows programmers to reify and generate undefined entities. However, Prorogued Programming requires that programmers explicitly mark what expressions will be prorogued concerns, *i.e.* undefined entities.

## IV. Conclusions

We exhibit that current IDEs cannot properly support incremental programming. As a solution, we present a simple concept, called Ghosts (available for Java and Smalltalk [8]). Ghosts are non-intrusive reifications of undefined entities that can be traced, structured and type checked on the fly. Therefore, programmers can focus on their main task, and when needed they can generate whole code skeletons for some or all ghost entities in the current project.

As a next step, we will work on both making ghosts more usable and compatible with current tools, such code completion, and evaluating its benefits through a controlled experiment.

## REFERENCES


[1] K. Beck, *Test-Driven Development: By Example*. Addison-Wesley, 2002.

[2] S. W. Ambler, "How agile are you? 2010 survey results," 2010. http://www.ambysoft.com/surveys/howAgileAreYou2010.html.

[3] Microsoft Developer Network, "Generate From Usage." http://msdn.microsoft.com/en-us/library/dd409796.aspx.

[4] DevExpress, "CodeRush – consume-first development." http://devexpress.com/Products/Visual_Studio_Add-in/Coding_Assistance/consume_first_development.xml.

[5] JetBrains Inc, "ReSharper plugin for VisualStudio." http://www.jetbrains.com/resharper/.

[6] JetBrains Inc, "Intellij idea." http://www.jetbrains.com/idea/.

[7] O. Callaú and É. Tanter, "Programming with ghosts," *IEEE Software*, vol. 30, no. 1, pp. 74–80, 2013.

[8] O. Callaú and É. Tanter, "Ghosts." http://pleiad.cl/ghosts.

[9] W. Miao and J. Siek, "Incremental type-checking for type-reflective metaprograms," in *Proceedings of the 9th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE 2010)*, (Eindhoven, The Netherlands), pp. 167–176, ACM Press, Oct. 2010.

[10] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, objects," in *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, (New York, NY, USA), pp. 236–246, ACM, 2004.

[11] M. Afshari, E. T. Barr, and Z. Su, "Liberating the programmer with prorogued programming," in *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, Onward! '12, (New York, NY, USA), pp. 11–26, ACM, 2012.