Lazy Types: Automating Dynamic Strategy Selection

Fernando Berzal, Juan-Carlos Cubero, Nicolás Marín, and María-Amparo Vila, *University of Granada*

The reflective technique of *lazy typing*—deferring the exact definition of object methods until the last possible moment—can help programmers more easily and consistently deal with partial or incomplete data.

Since the early days of structured methodologies, design techniques have evolved to facilitate the representation of real-world entities in software systems. Although object-oriented modeling and design techniques have made developing complex applications easier, many applications deal with data and behavioral requirements that conventional design models have difficulty accommodating.

Fortunately, software design techniques didn't stop evolving after object orientation's introduction. Years ago, Karl Lieberherr advocated *adaptive*, or *structure shy*, programming. This object-oriented programming (OOP) extension attempts to bind algorithms to data structures as late as possible to maximize the software's modularity, purportedly making software easier to understand and maintain.[1] (Lieberherr's Demeter Project met with limited success because it required using modified programming languages.) From a more database-oriented perspective, semistructured data management systems let programmers represent data whose structure is not completely regular.[2] Here, the ubiquitous (and often hyped) XML-based technologies,[3] in particular, have induced a new wave of advances.

We introduce a novel OOP extension, merging both philosophies (adaptive programming and semistructured data management), that you can use transparently in current programming platforms. *Lazy types* model variability without increasing implementation complexity. In situations where the use of standard design patterns could hinder design understandability, lazy types deal with complexity by keeping track of a unique type while avoiding the artificial multiplication of types traditional OOP solutions introduce. Moreover, lazy types reduce potential error sources as well as the amount of code programmers must write.

A case study

We can use existing OO models to represent complex data and behavior, but the data's structure (and, hence, its behavior) must stay the same. Yet, many applications must deal with data that doesn't easily fit into static models, and the data can present structural ir-

0740-7459/05/$20.00 © 2005 IEEE

regularities (for example, data collected from data sources with different schemas).

So, the objects that make up an OO system at runtime might need to act on varying amounts and kinds of data. Methods might need alternative implementations to account for this variance. All in all, program behavior should adapt itself to the data available at each moment.

We can use standard class hierarchies and strategies to handle these cases, but they can become too complex and difficult to maintain given the conditional logic needed to dynamically adjust alternative implementation strategies.

For example, suppose that we need to develop a software system that will provide up-to-date information about the farms in a certain geographical region. The system will provide information such as the area for each plot of land devoted to a single crop and the number of trees in each plot. Our system will also provide aerial images of the farms for inspection. Government agencies find this kind of information to be extremely useful because they usually need to forecast annual production, measure crop rotation, estimate losses due to natural disasters, or budget funds for subsidizing farming activities.

However, the information available for each plot at any moment might vary. There might be some basic registry data for all plots, aerial images for some, and detailed historical records for only a small fraction of them. In such situations, when we want to compute a given plot's area, the available information might range from a rough location of the plot to its exact perimeter. Suppose the property registry provides its actual area so that we don't need to estimate it. We could approximate the plot's number of trees by using the plot area and the average tree density for a certain kind of crop. We could also automatically compute that number from an aerial image once we know the plot's geographical limits. In short, we have alternative ways to compute what we're interested in, sometimes with differing degrees of precision, sometimes from different data sources.

Then again, data might not always arrive in the same order nor be available indefinitely. For instance, a survey process providing information on plots might be at different stages for different geographical zones. The collected information might also become outdated with time. These dynamic changes make our problem even harder.

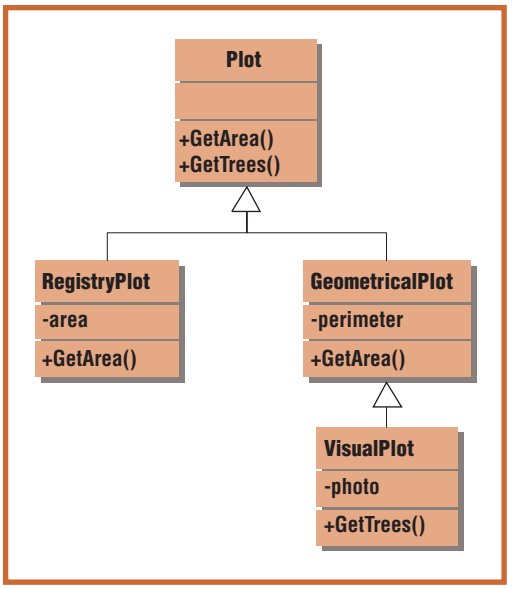

Figure 1. A class hierarchy for the plot problem.

Conventional solutions

When we face a problem such as our case study, conventional OO design techniques (class hierarchies and the strategy design pattern, in particular) provide potential solutions.

Class hierarchies

We can create a class hierarchy and override method implementations when appropriate, thus providing the needed polymorphism. In our example, an abstract class might contain an implementation of the `GetTrees()` method that would estimate the number of trees from the plot area. However, we could compute this area in several other ways depending on the available information. Subclasses of the base abstract class would be responsible for these alternative implementations. If we had actual images of the plot, we could also override the `GetTrees()` method of the `Plot` class and analyze those images by using image morphology techniques to obtain the actual number of trees in the plot (see figure 1).

The base classes in a class hierarchy could become interfaces if they were completely abstract. In our example, the `Plot` class would just define the common interface for all its subclasses in the hierarchy.

The use of class inheritance and polymorphism lets us manage alternative implementa-

```
public float GetArea() {

   if ( isActualAreaAvailable() )
     return area;
   else if ( isPerimeterAvailable() )
     return GetAreaFromPerimeter();
   else
     return UNKNOWN_AREA;
}

public float GetTrees() {

   if ( isAerialImageAvailable() )
     return EstimateTreesFromPhoto();
   else if ( !unknownArea() && isCropKnown() )
     return GetArea() * AverageTreeDensityForCrop();
   else
     return UNKNOWN_NUMBER_OF_TREES;
}
```

(a)

```
public float GetArea() {
  return areaStrategy.GetArea(this);
}

public float GetTrees() {
  return treeCountStrategy.GetTrees(this);
}
```

(b)

Figure 2. (a) A single monolithic `Plot` class would contain the conditional logic needed to select a suitable implementation for the `GetArea()` and `GetTrees()` methods. (b) The implementation of the `GetArea()` and `GetTrees()` methods using strategies, which must be updated whenever the object state changes.

tions for `GetArea()` and `GetTrees()`. However, even small changes in our understanding of the problem domain could cause huge reorganizations of the class hierarchy.

In our naive example, we were luckily able to put all of the logic into the class implementations without having to duplicate any code. However, it's more likely that no simple inheritance hierarchy will fit our needs. For example, we could compute the number of trees from the aerial image and prefer to rely on registry data about the plot area, a situation that's beyond the scope of the class hierarchy in figure 1.

Moreover, because most programming languages today only accept single-implementation inheritance, a carefully crafted class hierarchy might easily become worthless when the design must adopt variability in several independent dimensions. Multiple-interface inheritance would be useful here from a declarative point of view, although it wouldn't avoid the need to implement the needed functionality variants, which would probably include a lot of duplicated code. (At the very least, if we properly avoided the proliferation of duplicated logic, the resulting code would include calls to the same methods time and again.)

A more subtle drawback of class hierarchies representing evolving objects appears when we obtain new data about a particular object. Then, the object might need to change its behavior and, hence, its type. Although the object would keep its external interface, the object type would change and the object identity would be lost with that change. In our example, that could happen if we receive an aerial image corresponding to a plot we had initially classified as a `RegistryPlot` instance. We might also need the inverse type migration, from `VisualPlot` to `RegistryPlot`, if the aerial image becomes outdated. Although such changes might seem reasonable at first, they are unsound if we must track an object's identity during its lifetime.

Strategies

When it's crucial to keep the object identity and when the problem domain gets so complicated that a class hierarchy becomes unmanageable, we can choose to not create such a hierarchy. We could include everything in a single monolithic class, including the conditional logic needed to select the suitable implementation for `GetArea()` and `GetTrees()` (see figure 2a).

This monolithic solution helps us keep object identities through their complete lifetime and avoid class hierarchy reorganizations during maintenance. However, the programmer must still maintain this poorly modularized code, including all the conditional logic needed for each method to choose the proper implementation depending on the current object state.

The well-known *strategy design pattern*[4] can help us properly modularize the solution to our problem. The strategy design pattern decouples the data an object encapsulates from the implementation of the algorithms that support the desired variability in object behavior. The resulting design would look like figure 3.

As the UML class diagram in figure 3

shows, we must add two new *attributes* (data fields) to the `Plot` class to keep the strategies responsible for implementing the `GetArea()` and `GetTrees()` methods. These methods' implementations are now trivial because they just delegate to the corresponding strategy (see figure 2b).

Unfortunately, those strategies must change when the object state changes. So, we must also include conditional logic to update the corresponding strategies every time the object state changes so that we invoke a suitable strategy given the available data. An ancillary `UpdateStrategies()` method could keep strategies up to date, but we should still remember to invoke that method every time the object state might change (for example, at the end of all setter methods). Every constructor should also specify a default strategy for each varying method.

It's obvious that the strategy design pattern elegantly solves some of the problems rigid inheritance hierarchies cause. However, it adds unnecessary implementation complexity and makes the programmer responsible for maintaining error-prone conditional logic to control the policy that determines which alternative implementation to use in each situation for each particular object.

Room for improvement

At first glance, we might use the previous solutions to solve our problem. However, they carry hidden costs that might emerge during system maintenance and evolution. When, as usually happens, the decision on what alternative strategy to use depends solely on the available information, previous solutions reflect implicit code duplication (that is, the conditional logic needed to set the ad hoc strategy). But we all know that we should always strive to avoid code duplication. This is where lazy types, which provide a transparent solution for our need of ad hoc polymorphism, are at their best.

Lazy types defined

Lazy types offer a more flexible solution for coping with the dynamic selection of alternative implementation strategies. Lazy types let objects change behavior at runtime. Lazy object behavior automatically changes when the object state changes. Additionally, lazy types easily fit into existing development practices.

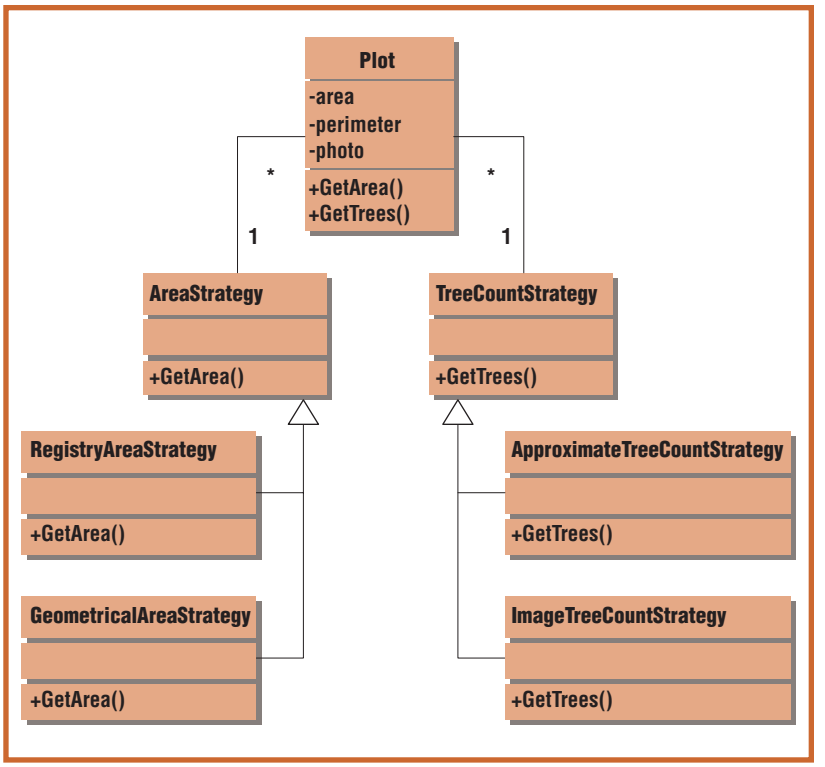

Figure 3. Use of strategies to solve the plot problem.

As we've seen, traditional OO designs work better when the data's structure doesn't vary. Irregular data handling in conventional OO modeling is difficult. It tends to add a lot of artificial complexity to the implementation of relatively common situations. When we need to manage entities with differing precision levels or when entities present structural irregularities, we require more expressive and powerful modeling techniques to concisely define the type of a given class of objects.

In conventional OOP, a type describes a set of objects equipped with certain operations. Classes are implementation modules that define types in OOP languages. Usually, a class sets up a single structure and behavior that are common to all its direct instances; you establish object structure and behavior beforehand.

By definition, a lazy type's structure and behavior dynamically adapt to the available data at the instance level. As with conventional types, a set of attributes determines its structure, and a set of method signatures defines its interface (see figure 4).

A *lazily typed object* (*lazy object*, for short) encapsulates a set of attributes. However, this set is not immutable in lazy objects, even though it's always a subset of the set of attributes defining the lazy type.

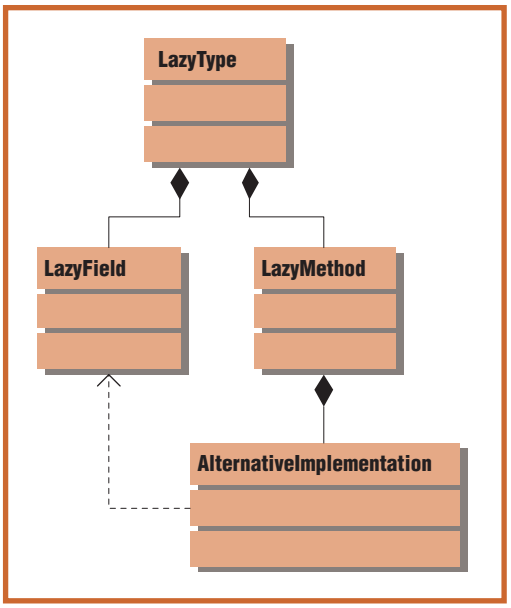

Figure 4. A lazy type includes a set of attributes and a set of methods, as usual in object-oriented programming. However, lazy-type behavior is described by means of alternative implementations for each method whose behavior dynamically changes depending on the available data.

A lazy object's structure doesn't have to perfectly fit its whole type definition. Some attributes might not always be present in a lazy object. The set of attributes dynamically evolves during a lazy object's lifetime.

Alternative method implementations describe lazy-type behavior for different structural situations. Due to these alternative method implementations, a lazy object's behavior can change dynamically depending on the available data. Alternative method implementations share their signature, so the type maintains its external interface and the programmer can transparently use lazy objects.

Invoking a lazy method will automatically delegate to one of the alternative implementations according to the object's current state. The implementation used will depend on the data each alternative method implementation needs. If a given attribute isn't available and a method implementation needs that attribute value, the lazy method won't use that alternative implementation. Instead, the lazy method will automatically invoke the alternative that best fits the current object state.

So, a lazy object will only incorporate the attributes it really needs at each moment and its behavior will change accordingly. In practice, you can automatically derive a lazy object's dynamic configuration from the source code of the alternative implementation strategies without requiring the programmer to add conditional logic or increasing the design model complexity.

Lazy types in practice

We recommend the following simple approach for defining lazy types in practice:

- Determine the set of methods that define the type interface. Because the interface must drive the implementation, and not the other way around, we start by focusing on the type interface.
- Discover the largest set of potential attributes for the lazy type (that is, all the attributes we think the type could ever have). This set of attributes becomes the lazy type's structural description.
- For each method included in the type interface, create the different alternatives that will implement the behavior associated with the method under different situations (that is, provide the alternative method implementations).

Once you've defined a lazy type, you can use lazily instantiated objects as standard objects in your programming language of choice.

Current programming platforms provide a defined method for adding declarative information to runtime entities such as classes, methods, and instance or class variables.[5] Metadata (whether attributes in .NET or annotations in Java) is stored with your program at compile time, so that you can retrieve and use it at runtime. This feature allows for the easy definition and transparent use of lazy types in standard programming languages.

Let's go back to our plot representation problem for a moment. First, we define the external plot interface—that is, the part of a plot behavior that stays the same regardless of its internal structure. In our example, the interface would at least include the `GetArea()` and `GetTrees()` methods. This interface becomes the lazy type's public interface.

Next, we identify all the data we might collect about a plot—namely, its observed area (`observedArea`), its geographical perimeter (`perimeter`), and an aerial photograph (`photo`).

Finally, we design alternative method implementations and implement the lazy `Lot` class that will replace our original `Plot` class. For instance, figure 5 shows how the lazy `Lot` class would look in C#. As you can see in the source code, a `[Lazy]` metadata attribute indicates that the class corresponds to a lazy type. An-

other attribute, `[AlternativeImplementation]`, marks the alternative method implementations that describe the lazy lot objects' dynamically varying behavior.

Our lazy class implementation in figure 5 looks like a standard class and, in fact, it can be unit tested as such with NUnit (www.nunit.org). However, it avoids the need to create a class hierarchy, and it lacks the artificial complexity of a strategy-based solution.

A reflective-object factory[4] creates lazy objects implementing the public lot interface. This factory permits the flexible instantiation of plots and their dynamic evolution. To create a lazy lot object, we would type

```
Lot lotObject = (Lot) LazyFactory.
   Create(typeof(Lot));
```

Once the lazy-object factory creates a lazy object, setting object properties will make strategies change automatically without programmer intervention. Figure 6 illustrates the evolution a lazy lot object might experience at runtime. You just invoke the object-published methods as usual, letting the underlying infrastructure select a suitable implementation alternative.

```
[Lazy]
 public class Lot {

  private float observedArea;
  private Polygon perimeter;
  private Image photo;
  …

  public float GetArea () {
     return observedArea;
  }

  [AlternativeImplementation("GetArea")]
  protected float GetAreaFromPerimeter () {
     return perimeter.GetArea();
  }

  public int GetTrees () {
     return (int) ( GetArea() * AverageTreeDensity );
  }

  [AlternativeImplementation("GetTrees")]
  protected int GetTreesFromPhoto () {
     return ImageMorphologyAnalyzer.GetObjectCount
        (photo, perimeter, AverageTreeSize);
  }
  …
 }
```

Figure 5. The lazy `Lot` class implemented in C# for the .NET Framework. A Java implementation would be similar. You would just need to use the Java annotations' syntax instead of the .NET attributes.

Implementation issues

We developed a proof-of-concept implementation of lazy types for the .NET Framework. Our generic library, freely available from http://elvex.ugr.es/software/lazy, allows the use of lazy types as described in this article.

Metadata enables the transparent use of lazy-typing capabilities in our applications. Metadata attributes tag lazy classes and define alternative method implementations. For instance, to support lazy types in the .NET Framework, we created the `[Lazy]` and `[AlternativeImplementation]` attributes in C# (see figure 7 on page 105).

Our implementation parses compiled intermediate code (Microsoft intermediate language, or MSIL, in the .NET Framework) to build a lazy-object model. We can then use a simple dataflow analysis of the alternative method implementations, as indicated by the metadata attributes, to determine when to invoke each alternative implementation. By using this information, implementation strategies can change accordingly in response to newly available data.

Our lazy-object factory dynamically creates new types to represent lazy objects using the reflection capabilities included in the .NET Framework. You can use *reflection*—an executing program's ability to examine itself—to discover metadata about types at runtime. Combined, metadata and reflection are extremely useful when we must perform tedious programming tasks.[6] In our context, reflection is essential when analyzing class structure and modifying object structure at runtime.

Even though our proof-of-concept implementation of lazy types uses reflective programming, we could have alternatively used metadata at compile time, following a code generation approach.[7] A preprocessor could create standard C# source code from the metadata-tagged lazy-type description. In programming languages without reflective capabilities, this would be our only choice for implementing lazy types.

Both styles of metadata-based programming help programmers avoid writing repetitive and error-prone code.

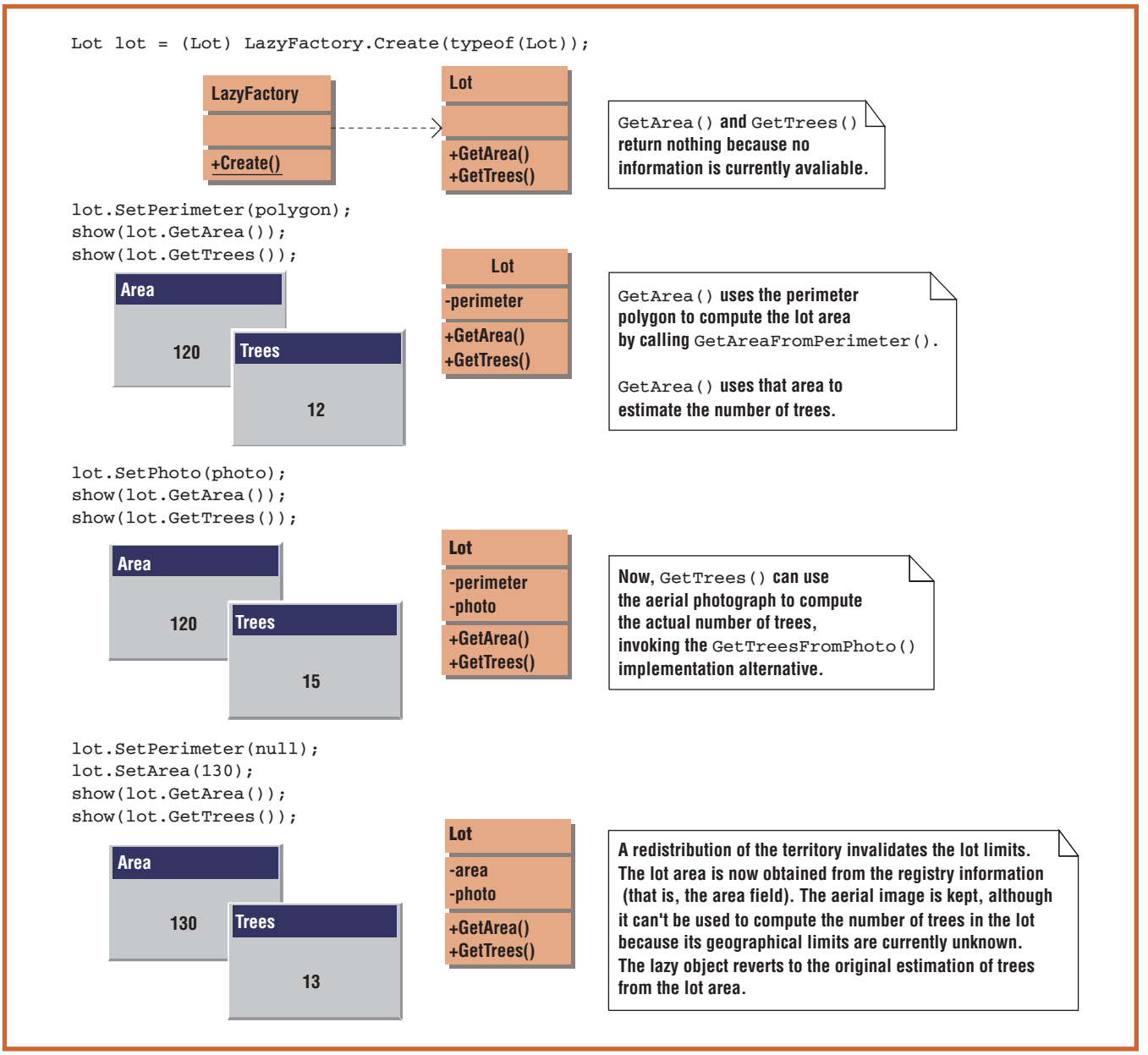

Figure 6. Evolution of a lazily typed lot object.

Figure 8 depicts the overall design of a lazy-typing framework for conventional programming platforms. The LazyFactory instantiates lazy objects. When the programmer invokes the LazyFactory.Create(type) method, the following steps take place:

- The `LazyFactory` builds a `LazyObjectModel` to describe the lazy type (including the `LazyMethodModels` and `LazyFieldModels` representing its methods and fields). This lazy-type model contains all the information needed to configure lazy objects dynamically, including data about alternative method implementations. In our implementation, we derive the `LazyObjectModel` at runtime directly from the MSIL code so that the original source code isn't necessary.
- The `LazyFactory` dynamically creates a new type (represented by a `LazyType` and its constituent `LazyMethods`) from the `LazyObjectModel`. Once we have the lazy type, we can instantiate objects belonging to this dynamically created class.

The resulting lazy object will automatically adapt its behavior to its current state during

```
[AttributeUsage(AttributeTargets.Class)]
public class LazyAttribute : Attribute {
}

[AttributeUsage(AttributeTargets.Method)]
public class AlternativeImplementationAttribute : Attribute {

   private string method;

   public AlternativeImplementationAttribute (string method) {
      this.method = method;
   }

   public string Method {
      get { return method; }
   }
}
```

Figure 7. The C# implementation of the `[Lazy]` and `[Alternative-Implementation]` attributes that support lazy types in the .NET Framework.

its entire lifetime. This exempts the programmer from having to write any conditional logic by hand.

Obviously, both the creation of the generic model for the lazy type and the creation of the new type derived from this model occur only once. At runtime, the `LazyFactory` stores generated models and types in an internal cache. This way, the instantiation of new lazy objects doesn't degrade application performance.

When many implementation alternatives are present for lazy methods, developers might have difficulty visualizing system behavior due to the sheer number of states the system can be in. This is true regardless of whether they use lazy types or the more conventional strategy design pattern. In the future, we plan to develop supporting tools to alleviate this problem.

In any case, lazy types reduce the amount of code a programmer must write when compared to previous alternatives. Additionally, lazy types reduce potential error sources because they fully automate the strategy selection process. Thus, lazy types make the resulting code less complex and easier to maintain.

As Eric Evans points out in *Domain-Driven Design: Tackling the Complexity in the Heart of Software*:

> *Domain models contain processes that are not technically motivated but actually meaningful in the problem domain. When alternative processes must be provided, the complexity of choosing the appropriate process combines with the complexity of the multiple processes themselves, and things get out of hand.*[8]

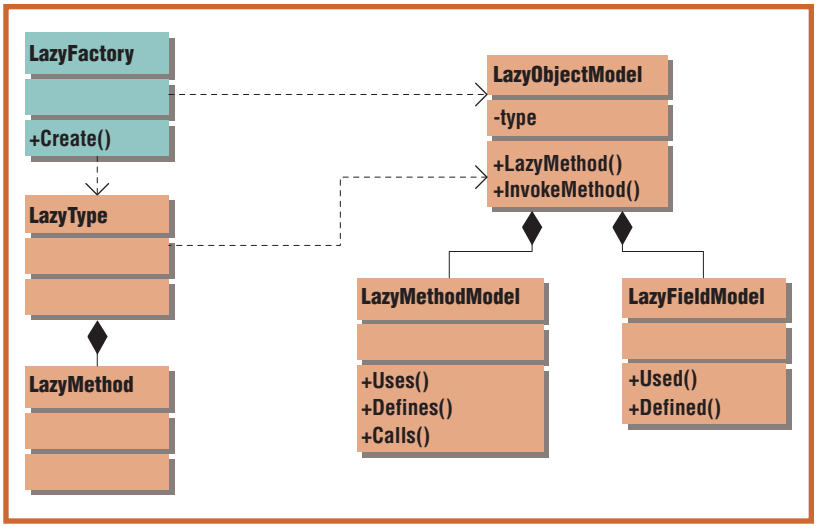

Figure 8. Lazy-object framework implementation details.

Multiple implementation strategies can represent those alternative processes. Whereas hand-coded conditional logic chooses among them, adding implementation complexity, lazy types eliminate the need to implement that choice explicitly, reducing the implementation complexity to the inherent complexity of the problem domain.

A general lazy-typing framework suffices to provide standard OOP platforms with the flexibility lazy types offer. Lazy types could be the link we need to seamlessly deal with semi-

About the Authors

Fernando Berzal is an assistant professor in the Department of Computer Science and Artificial Intelligence at the University of Granada. During the early preparation of this manuscript, he was a visiting research scientist at the University of Illinois at Urbana-Champaign. His research interests include model-driven software development, software design, and software mining. He received his PhD in computer science from the University of Granada. He is a member of the ACM, IEEE, and IEEE Computer Society. Contact him at Dept. of Computer Science and AI, Office 17, ETS Ingeniería Informática, Univ. of Granada, Granada 18071, Spain; berzal@acm.org.

Juan-Carlos Cubero is an associate professor in the Department of Computer Science and Artificial Intelligence at the University of Granada, where he coordinates the PhD program in intelligent systems. His research interests include database design, data mining, and software modeling. He received his PhD in computer science from the University of Granada. Contact him at Dept. of Computer Science and AI, Office 37, ETS Ingeniería Informática, Univ. of Granada, Granada 18071 Spain; JC.Cubero@decsai.ugr.es.

Nicolás Marín is a lecturer in the Department of Computer Science and Artificial Intelligence at the University of Granada and a researcher in the Intelligent Databases and Information Systems Research Group. His research interests include database design, data mining, software modeling, and mathematical theory. He received his PhD in computer science from the University of Granada. He is a member of the IEEE Computer Society. Contact him at Dept. of Computer Science and AI, Office 17, ETS Ingeniería Informática, Univ. of Granada, Granada 18071, Spain; nicm@decsai.ugr.es.

María-Amparo Vila is a full professor in the Department of Computer Science and Artificial Intelligence at the University of Granada, where she leads the Intelligent Databases and Information Systems Research Group. Her research interests include database design, data mining, and mathematical theory. She received her PhD in mathematics from the University of Granada. Contact her at Dept. of Computer Science and AI, Office 38, ETS Ingeniería Informática, Univ. of Granada, Granada 18071, Spain; vila@decsai.ugr.es.

structured data in conventional software development environments. At least, we believe they're a first step in the right direction.

Acknowledgments

Spain's Interdepartmental Commission on Science and Technology (Comisión Interministerial de Ciencia y Tecnología) partially supported this work under grants TIC2003-08687-C02-02 and TIC2002-04021-C02-02. The Andalusian regional government (Junta de Andalucía) supported Fernando Berzal's stay with Jiawei Han's research group at Urbana-Champaign.

References

1. K.J. Lieberherr and I. Holland, "Assuring Good Style for Object-Oriented Programs," *IEEE Software*, Sept./Oct. 1989, pp. 38–48.
2. S. Abiteboul, P. Buneman, and D. Suciu, *Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kaufmann, 1999.
3. B. Benz and J.R. Durant, *XML Programming Bible*, John Wiley & Sons, 2003.
4. E. Gamma et al., *Design Patterns*, Addison-Wesley, 1995.
5. J. Newkirk and A.A. Vorontsov, "How .NET's Customs Attributes Affect Design," *IEEE Software*, Sept./Oct. 2002, pp. 18–20.
6. M. Fowler, "Using Metadata," *IEEE Software*, Nov./Dec. 2002, pp. 13–17.
7. J. Herrington, *Code Generation in Action*, Manning Publications, 2003.
8. E. Evans, *Domain-Driven Design: Tackling the Complexity in the Heart of Software*, Addison-Wesley, 2004.