Compilation of Prototype Objects into Class Objects Using Profile Information

Lawrence J. Karnowski and Bradley T. Vander Zanden
University of Tennessee
{karnowsk, bvz}@cs.utk.edu

Abstract

The prototype-instance inheritance model has a high storage cost. This storage requirement often forces applications into virtual memory, significantly impairing their interactive performance. We believe this overhead can be reduced sufficiently to avoid using virtual memory. The key observation is that large applications have relatively few prototype objects, each of which is replicated very frequently. Once the design is stabilized, these prototypes do not change. Consequently, they can be compiled into classes. This paper describes a scheme for performing such compilations based on profile information. The scheme provides a way to convert a class-instance object to a prototype-instance object if the profile information is incomplete or inaccurate. It also lays the groundwork for future storage optimizations on the compiled objects. This approach allows the software developer to benefit from prototype-instance inheritance during development and then move the prototype code directly to production.

Keywords: Language Design and Implementation, Programming Environments, Prototype-Instance Model, Storage Optimization, Graphical Interface Toolkits, Glyph Objects, Profile-Based Compilation, Transmogrification

Paper Type: Research

1. INTRODUCTION

The prototype-instance model [2, 17, 18, 21] provides a great deal of flexibility. Two of its most salient features are that: 1) any object can be made a prototype by instancing that object; the instance will inherit the prototype's data and method attributes and any changes in the prototype will be also reflected in the instance, and 2) any object can add or delete data fields and methods dynamically. These features allow the developer to alter the attributes and behavior of base prototypes at runtime, quickly making large changes throughout the entire system. This functionality readily supports rapid-prototyping.

However, this flexibility is very expensive in terms of memory use. The ability to dynamically add and delete data and method attributes (called *slots* [20]) incurs a large amount of memory overhead. For example, an application implemented in the Amulet toolkit [20], a prototype-instance toolkit used for quickly developing graphical user interfaces, will run into virtual memory after creating only a few thousand objects [11]. Accessing virtual memory creates unacceptable pauses in the application's interactivity.

Does this mean that prototype-instance inheritance must be abandoned to develop a production-quality interface for an application involving millions of objects? Intuitively it seems unlikely that each object among millions will have a completely unique type (i.e., unique set of slots). It seems more likely that there will be a few prototypes, each defining a unique type and having thousands of instances. We predict that a relatively small number of types will be used by any large interface. Using type information, we can compile these types into less customizable but also less memory-expensive classes. Ultimately, using these more space-efficient classes, interfaces consisting of millions of objects will be possible.

This paper describes our first step towards achieving the goal of less memory-expensive objects--a novel framework that allows an interface to mix prototype-instance objects with class-instance versions of those objects. The interface developer writes and debugs an application using a prototype-instance based toolkit. The developer then executes the application with a profiler. The profiler determines the unique types in the application. This information is given to a code generator that creates classes from the type information. The application is then recompiled with the new classes. The developer never has to rewrite a single line of code.

The rest of this paper is organized as follows. The next section discusses related work. The third section gives an overview of the prototype-instance inheritance system as it is implemented in Amulet, the toolkit used in our experimental work. The fourth section describes our class-instance replacement scheme. The fifth section provides information about the implementation of this scheme. The sixth section documents our empirical results. The seventh section describes the current implementation and discusses future work. The last section concludes.

2. RELATED WORK

Related research falls into two categories: profile-based compilation and existing optimizations.

2.1 Profile-Based Compilation

In the past few years, researchers have successfully exploited dynamic profiling as a means of improving the performance of prototype-instance systems without sacrificing their ease of development. Dynamic profiling obtains information that can be used for both static optimizations and for dynamic compilation of frequently executed portions of a program. Both the SELF [4, 5, 14] and Cecil [6, 10] languages have successfully used this approach.

Profile-based compilation automatically instruments a program's code and, based on the information collected, dynamically optimizes frequently executed portions of the code. These code segments are optimized using techniques such as type prediction, method inlining, and code splitting [4, 22]. Profiling and dynamic compilation have allowed the performance of object-oriented languages to come within a factor of 2 of C programs [5, 13]. In addition, they do not adversely affect interactive performance [14]. Finally, a variety of studies have shown that profiling is applicable for 1) programs under rapid, iterative development, 2) programs handling input sets that differ from the training sets, and 3) programs involving graphical interfaces [24, 10].

In this paper, we describe how a profiler can be used to obtain type information that can then be used to statically compile prototypes into classes. The profiler is invoked as the application is shutting down. Consequently, the profiler has no effect on interactive performance since it becomes active only at the termination of the program.

Another hallmark of dynamic compilation is that it gracefully handles situations for which no optimized code exists by falling back on less optimized code or an interpreter. For example, a certain spot in the code may have been optimized for integer operands (e.g., the plus operator may have been inlined). If operands of an unexpected type occur, a method is invoked that is capable of handling unexpected types. By invoking a method, rather than using inlined code, the operation will take longer, but it will still perform correctly. Similarly, the scheme described in this paper gracefully handles situations in which the profile information is incorrect or incomplete. Specifically, it will convert a class-instance object back to a prototype-instance object and forward all future requests to the new object.

2.2 Storage Optimizations

A number of storage optimizations have been suggested for object models, including glyphs [3] and virtual aggregates [19]. Glyphs are extremely lightweight objects that contain only essential information about an object that cannot be computed by other means [3]. For example, a label object in a list might contain the value of the string, but not its x or y coordinates, if these coordinates could be computed externally (e.g., by constraints or by the composite list object). The advantage of glyphs is that they provide the same interface that a typical structured graphics object provides, without the corresponding space overhead. The class compilation scheme described in this paper is intended to lay the foundation for automatically compiling prototype objects into glyph-like objects. Previously programmers have had to manually handcraft the glyph objects.

Virtual aggregates are composite objects that give the illusion of having internal parts, but do not represent these internal parts explicitly [19]. Virtual aggregates are typically formatting objects, such as lists or tables, that position a set of homogenous objects. If an operation needs one of the objects, the virtual aggregate creates temporary objects for the operation "on demand". Once the object is no longer needed, its space is reclaimed. The class compilation scheme described in this paper provides an attractive vehicle for implementing a virtual aggregate strategy. In particular, a profiler could be tuned to collect information about types of objects that could benefit from virtual aggregates (i.e., objects with lots of homogenous parts, none of which are frequently requested). These objects would be compiled into virtual aggregates, and they would generate part objects on demand.

Constraints are an increasingly common component of object systems [18, 15, 12, 1, 8, 20] and a number of researchers have tried to minimize the storage costs of constraints, including Freeman-Benson's module approach [7], Hudson's micro-constraints [16], and Halterman's model dependencies [11]. The module approach attempts to eliminate constraints by compiling them into plans. The micro-constraints approach employs a RISC-like strategy. It defines a few common layout constraints, specifies what types of operands they may have, and represents them in 32-bit words. A few bits specify the type of constraint, a few bits the operands, and a few bits a possible constant value. The model dependencies approach represents constraint graphs using implicit dependencies and generates explicit dependencies on demand. For each prototype object, a model dependency graph is constructed for that object. Instance objects use this model dependency graph to find the constraints that are affected by changing one of the slots in the instance object. The model dependency graph is similar to our approach in that it takes advantage of type information. However, it is more of a dynamic compilation approach since it dynamically computes and installs model dependency graphs. It also does not depend on profile information.

3. OVERVIEW OF AMULET PROTOTYPE-INSTANCE SYSTEM

This section first briefly contrasts prototype-instance inheritance with class-instance inheritance. It then describes features of the Amulet toolkit for those who are not familiar with it.

3.1 Prototype-Instance Inheritance

In a class-instance inheritance model, there is a distinction between object types and object instances. An object type is similar to a class definition in C++ or Java. It defines a precise interface of data members and behavior methods. An object type cannot be altered at runtime, and it is not allocated memory at run-time (i.e., it is not an object). The object instance is the embodiment of an object type in executable format. It is allocated space, stores data, and its methods can be executed.

In a prototype-instance inheritance model, the distinction between object type and object instance is blurred. Here there is only one type of object, and it has a dual nature. First, each object is an object instance. It is allocated memory and it stores data and methods in a set of attributes called *slots*. There is no distinction between data and methods. A slot can store either one. Second, each object is an object type. It has a Create method that allocates memory for a new object, initializes it for use, and then returns it. The Create method allows an object to become a prototype by creating instances of it. Initially, the new instances inherit the slot set and values of their prototype, but each object can dynamically change the values of its slots (including its method slots) or add or delete slots. Hence objects can dynamically change their interface.

3.2 Amulet Features

Amulet is a prototype-instance-based toolkit that provides several features that greatly facilitate rapid-prototyping of graphical user interfaces. These features include 1) composite objects, 2) constraints, and 3) triggers.

A *composite object* is an object that aggregates and organizes other objects [9]. A composite object is treated like a primitive object. For example, a developer can create instances of a composite object, move a composite object, and delete a composite object. Composition is highly useful in a graphical user interface. For example, if a developer wanted to create an interface that relied heavily on displaying smiley faces, it would be annoying to have to create a new smiley face object from scratch each time. Also, the developer would not want to duplicate code when creating, deleting, or moving a smiley face. Composition eliminates this wasted effort. The developer would create a SmileyFace composite object that contained a yellow circle as the face, two black circles as the eyes, and one black arc for the mouth. Each time a smiley face was needed, an instance of a SmileyFace object would be created.

A *constraint* is a relationship between two values that is expressed as a mathematical formula and is then maintained automatically by the system [23]. Constraints can be used for graphical formatting, or for automatic value updates like a cell in a spreadsheet. For example, a developer would use constraints to automatically compute the size of the eyes and mouth inside the SmileyFace object. When the composite object changed size, all the interior objects would also automatically change size proportionally.

Triggers (or as Amulet calls them, *demons* [20]) are behaviors contingent upon a slot change. When the slot changes, the trigger is automatically executed. Triggers facilitate the implementation of constraints ("when value A changes update values B and C"), and automatic functionality ("when the window closes, close all open files").

3.3 Anatomy of an Amulet Object

Amulet is implemented in C++. Consequently, all Amulet objects must be instances of some class, and this class is called Amulet_Object. Amulet_Object contains a variety of instance variables that support prototype-instance inheritance, including the management of instances, slots, composition, and triggers. The composition of an Amulet_Object is shown in Table 1. The storage cost of an Amulet object, exclusive of storage for slots, is 48 bytes.

Feature	**Number of Bytes**	**Variable Description**
Inheritance	12	Pointers to an object's prototype and to the object's list of instances.
Slots	4	A pointer to a dynamic array of slot objects
Composition	16	Pointers to an object's owner, its list of parts, and its part slot (the object's owner has a pointer to this part slot that allows the owner to directly access this object)
Triggers	16	Storage for a pointer to an object's set of triggers, a pointer to a queue for queuing the triggers for execution, and a number of fields for handling the inheritance of triggers.

Table 1: A list of instance variables in an Amulet_Object and the features that these variables support.

Each slot in an Amulet object is an instance of a class called Amulet_Slot. A slot object contains instance variables for handling triggers, constraints, and a number of housekeeping functions. A slot object consumes a minimum of 28 bytes, excluding storage space for a value and several lists required for constraint maintenance. This paper considers only the compilation of prototype objects into class objects, since creating the class objects is a prerequisite for compiling the slot objects. However, in the future we hope to extend this scheme to slot objects as well.

An average Amulet object contains roughly 40 slots and typically consumes well in excess of 1,000 bytes of storage. Consequently, an Amulet application often moves into virtual storage after only a few thousand objects are created. At this point interactive performance degrades significantly as the disk is repeatedly hit.

3.4 Exploiting Types

As noted in the introduction, most interfaces that create large numbers of objects use only a few distinct prototypes (types). These objects inherit the same triggers and slots. For example, it is very uncommon to change the set of triggers that are associated with an object. Similarly, it is unusual to change the methods that an object inherits. Consequently, much of the information for triggers and slots can be stored in a prototype object and inherited by each of the instances.

We can exploit this insight by compiling prototype objects into classes and creating instances of these classes at run-time. Trigger information and many slots can be stored as class variables, with only slots whose values are frequently altered by instances actually stored in the instance objects.

4. DESCRIPTION OF THE CLASS COMPILATION SCHEME

This section describes the design of the class compilation scheme and the next section describes its implementation

4.1 Scheme Overview

The class compilation scheme is predicated on profiling. A developer executes an application using a script of representative operations that might be performed by an end-user. When the application is terminated, the run-time environment invokes a profiler. The profiler examines each of the objects in the prototype-instance hierarchy, beginning at the root. The profiler examines each of the slots in a prototype object and determines whether the slot should be treated as a class variable or an instance variable. The test is based on a certain percentage of the instances either inheriting the slot or treating it as a local variable. Currently a slot is classified as a class variable if 80% or more of the instances inherit it; otherwise it is classified as an instance variable. The profiler writes the information about each of the prototype's slots to a data file.

A code generator reads the data file and generates a class for each of the prototypes listed in the file. Class slots are declared as static instance variables. Instance slots are handled by declaring an array of slot objects equal in size to the number of instance slots. Each instance of the class will allocate storage for this array. Each class also declares a slot accessor method called *find_slot* that takes a slot key as a parameter and returns the associated slot object. A read method can then return the slot's value, and a write method can set the slot's value. A slot key is an integer key that denotes a slot. For example, LEFT might be assigned the key 100 and TOP might be assigned the key 101.

The new class-instance objects are called *lightweight objects*, to distinguish them from the regular prototype-instance objects that are called *heavyweight objects*. The lightweight objects cannot dynamically add or delete slots, but they also do not suffer from the overhead of dynamic slot sets.

The lightweight objects replace the heavyweight objects in the application. However, the application should still have the illusion that a prototype-instance system is being used. In particular, if the profile information is incorrect or incomplete, it should still be possible to change the set of slots associated with an object. For example, the application might display an error dialog when a disk runs out of space. This condition might not occur during the profiling of the application, so the dialog window might try to add a slot to a lightweight object. A lightweight object will be too simple to handle conditions for which it was not designed, like adding a new slot. In this case the lightweight object will be *transmogrified*, or changed, to an equivalent heavyweight object. The slots will be copied to the new heavyweight object and the requested operation will be forwarded to the heavyweight object. The new heavyweight replacement will persist for the lifetime of the application.

4.2 Old Versus New Architecture of Amulet

To allow lightweight objects to replace heavyweight objects, and also to allow transmogrifications, several design changes had to be made to the Amulet toolkit. The original Amulet system used a Proxy pattern [9] in its object implementation. The Amulet_Object class was actually a wrapper class that contained a pointer to an implementation object (Figure 1.a). Amulet_Object was the proxy that was exported to the user to allow reference counting and to prevent a user from obtaining a direct pointer to the implementation object. However, multiple Amulet_Objects could share pointers to the same implementation object.

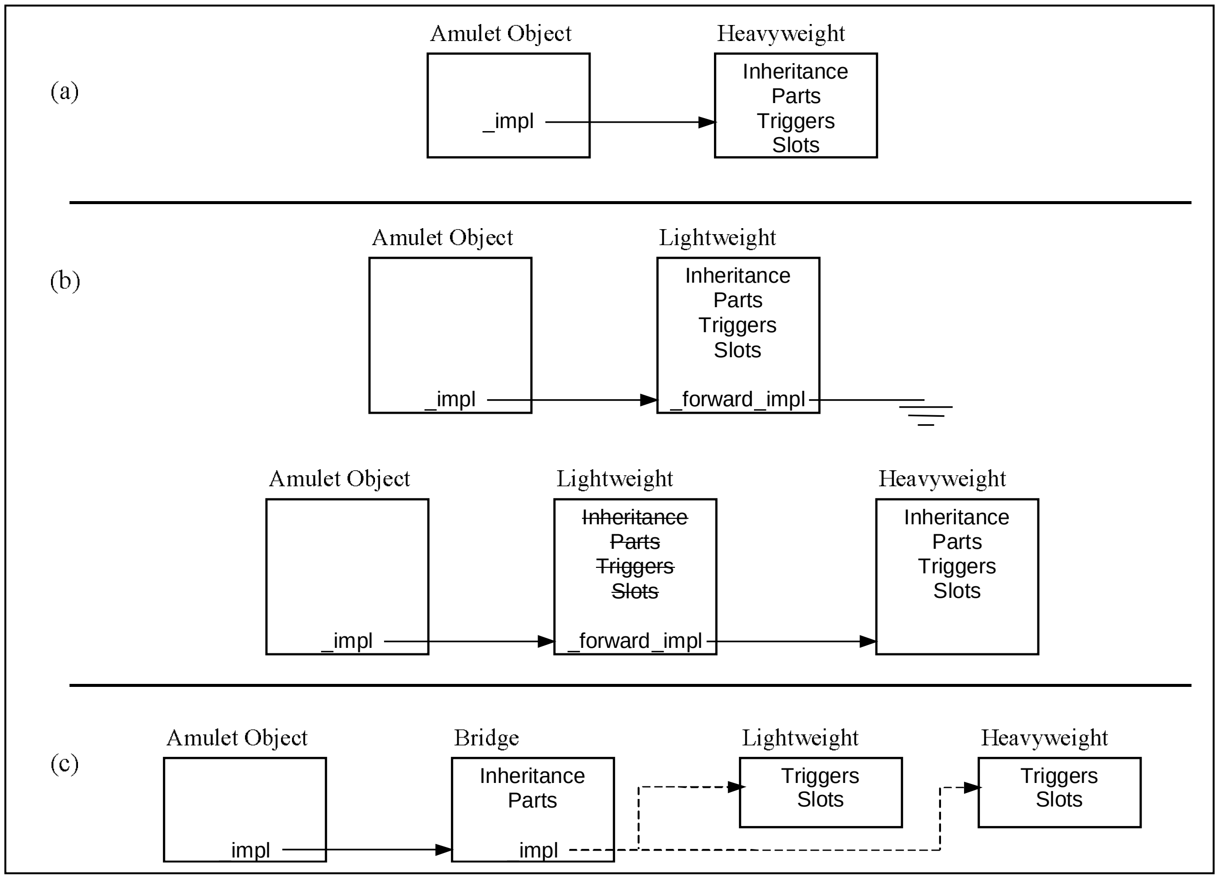

Figure 1: (a) The original implementation of an Amulet object. (b) An implementation of the class compilation scheme using a proxy pattern. (c) The final implementation of an Amulet object using a bridge pattern. `_impl` can point to either a lightweight or heavyweight object.

Our first idea was to allow an Amulet_Object to point to a lightweight object instead of a heavyweight object. If the lightweight object received an operation that it could not handle, it would create a heavyweight object and save a pointer to it (Figure 1.b). Thereafter, any requests received by the lightweight object would be forwarded to the heavyweight object. This scheme required that the lightweight object be retained. However, since few transmogrifications are expected to occur, the storage overhead of this decision was minimal. However, this idea foundered when we discovered that some of the functionality associated with composition and inheritance still had to be handled by the lightweight object. Consequently, both the lightweight object and the heavyweight object had to share some similar state. This arrangement proved awkward.

Consequently, we changed the design so that a lightweight object could be completely replaced with a heavyweight object. To accomplish this task, we changed the implementation object so that it used a bridge pattern rather than a proxy pattern. The new model still has an Amulet_Object class containing a pointer to an implementation class. However, this implementation class in turn has a pointer to a lightweight or heavyweight data object (Figure 1.c). The implementation class handles instance and part management, and the lightweight/heavyweight data class handles all trigger functionality and the storing and accessing of slots. When an error occurs and we must transmogrify the object, we simply replace the lightweight data object with a heavyweight data object with equivalently valued slots.

4.3 Details of Lightweight Implementation

This section gives a more in-depth look at the internals of a lightweight object. The layout of a lightweight class is shown in Table 2 and the layout of a lightweight subclass is shown in Table 3. The lightweight class is an abstract base class. It handles the basic functionality of lightweight objects, including transmogrification, manipulations of slot objects and triggers, and the storing of trigger information. Each of the lightweight subclasses implements a specific type. Each subclass stores the slot objects and provides methods for creating a concrete instance and accessing and initializing the object's slots. The layout of each of these objects is described in greater detail below.

4.3.1 Lightweight Class

As shown in Table 2, the lightweight class is responsible for implementing the operations declared in the Amulet interface and for handling transmogrifications. It also stores the object's trigger information. Both lightweight and heavyweight classes inherit from the same abstract base class. A lightweight class supports many, but not all of the operations declared in the Amulet interface. When it does support an operation, the operation can often be implemented more simply than in the heavyweight version, because a lightweight object does not have to propagate the changes to any instances (a lightweight object is always a leaf in the prototype-instance hierarchy).

For operations that a lightweight class does not support, such as remove slot, create instance, or changing the value of a class slot, the lightweight methods call a transmogrify method. This method creates a new heavyweight object, copies the lightweight object's trigger and slot information to the heavyweight object, and makes the implementation object point to the new heavyweight object. The lightweight object's method then forwards the request to the heavyweight object's method.

Section	Name	Purpose
Instance Variables	Trigger Information	A set of instance variables that store information about triggers, such as the set of triggers associated with this object, the list used to queue the triggers for execution, and the set of triggers that each slot inherits by default. The trigger information is stored in the lightweight class, rather than in the implementation object (i.e., the bridge object), because it is tightly connected with slot objects.
Methods	Transmogrify	Allocate storage for a heavyweight object, copy the trigger and slot information to the new object, and return a pointer to the new object.
	Methods that implement the Amulet interface	These methods are implemented separately in lightweight and heavyweight classes, since lightweight objects are always leaves in the prototype-instance hierarchy. So, unlike heavyweight objects, they never have to propagate information to instances, and thus can be implemented more simply than the corresponding heavyweight methods.

Table 2: The layout of a lightweight class.

Ordinarily the lightweight object would destroy itself once the operation is complete. However, our current implementation does not actually copy the slot information to the heavyweight object but instead makes the heavyweight object point back to the slots in the lightweight object. Consequently the lightweight object is retained, although it is no longer referenced. Since transmogrifications are infrequent, the extra storage overhead incurred by this decision is minimal.

4.3.2 Lightweight Subclasses (Prototype Classes)

Each lightweight subclass corresponds to a prototype object in the prototype-instance hierarchy. It provides storage for an instance object's slots, a method for retrieving a slot object given a slot key, and methods for creating an instance object and instantiating its slots. Slot storage and retrieval is described in this section. Section 5.3 describes slot initialization.

Slots are partitioned into instance and class slots. An instance slot has a changeable value and needs storage in each instance of the class. An example of instance slots would be the x and y position of the object on the screen.

Class slots do not have changeable values. They are initialized once and then never changed. Instead of making each instance of the lightweight object waste space in storing this constant-valued slot, we only store one class slot for each lightweight class. Each instance of the lightweight object has access to it, but none have to store it. A good example of this type of slot is the draw method. In Amulet, to encapsulate drawing behavior, each object has a slot that stores a pointer to the method that draws it likeness on the screen. It is obvious that the value of this slot will be the same for each instance of the lightweight object, and that the value will never change. Accordingly, this slot is made a class slot.

Section	Name	Purpose
Class Variables	Class slots	One slot object is allocated for each class slot
	Instance_Keys_Array	An array of slot keys that identifies each of the instance slots
	Class_Keys_Array	An array of slot keys that identify each of the class slots
Instance Variables	Slot_Array	An array of slot objects. There is one slot object for each instance slot.
Class Methods	Create	The create method called by a prototype in order to generate a lightweight instance.
	Initialize_Class_Slots	A static method called by the create method the first time the prototype creates an instance. This method instantiates each class slot by copying the contents of the corresponding slot object from the prototype object to the class slot.
Virtual Methods	Find_Slot	A method that takes a slot key and then returns the corresponding slot object.
	Initialize_Instance_Slots	A method that instantiates each instance slot by copying the contents of the corresponding slot object from the prototype object to the instance slot.
	Slot_Iterator	Returns an iterator object. The iterator object returns each of the slot keys in the class (both the instance slot keys and the class slot keys)

Table 3: The layout of a lightweight subclass.

The lightweight class hierarchy is only one level deep (Figure 2). When a lightweight class is created for a prototype, all the slots from both the prototype and its ancestors are included in the lightweight class. The original implementation of lightweight classes mirrored the prototype-instance hierarchy with lightweight classes subclassing other lightweight classes. However, this multi-level hierarchy merely complicated the implementation without providing any real benefit, so it was scrapped and replaced with the single-level hierarchy.

A find_slot method is implemented for each lightweight subclass. Amulet uses a *find_slot* method to find and return a slot object. Slot operations can then perform operations on the returned slot, such as reading or writing its value. In a heavyweight object, the slot objects are stored in a dynamic slot array. Amulet uses a linear search to locate a slot object. If the slot object cannot be found in the object's array, the prototype object's array is searched, and so on until the slot object is located or the root of the inheritance hierarchy is reached.

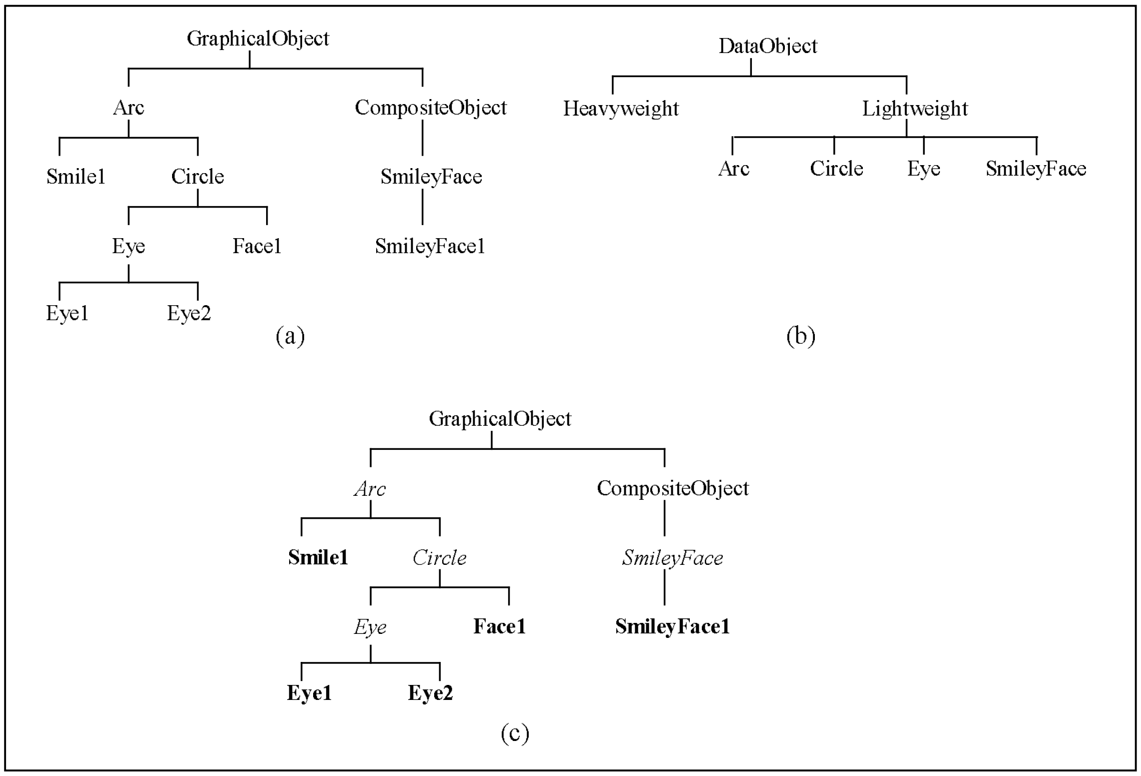

Figure 2: A sample prototype instance hierarchy is shown in (a). This hierarchy is compiled into the class-instance hierarchy shown in (b). CompositeObject and GraphicalObject are not compiled into classes because they do not have any leaf instances and hence will not generate any lightweight objects. (c) shows the prototype hierarchy that results with compiled class objects. The objects in normal type are heavyweight objects, the objects in italics are heavyweight objects that can generate lightweight objects, and the objects in boldface are lightweight instance objects.

In contrast, each lightweight class defines its own *find_slot* method. The method is simply a case statement that takes a slot key as input and returns the appropriate slot object from either the instance slot array or the appropriate class slot object. The switch statement identifies the key in O(1) time, whereas the linear search required by the heavyweight implementation requires O(n) time, plus a possible search through multiple levels of the prototype-instance hierarchy.

Each lightweight class keeps static arrays for class and instance keys. These arrays are necessary for initialization of the object, slot iteration, and other internal housekeeping routines.

5. IMPLEMENTATION SPECIFICS

This section gives details about other implementation specifics necessary to our optimization but external to the actual class compilation system. This includes profiling and object creation.

5.1 Profiling

As noted earlier, the profiler examines each of the prototypes in the prototype-instance hierarchy. Amulet provides a mechanism for traversing the slots in an object. The profiler uses this mechanism to examine each of the prototype's slots (the slot traversing mechanism returns both slots stored locally in the prototype and slots stored in any of the prototype's ancestors; hence both inherited and non-inherited slots are examined). For each slot, the profiler examines the same slot in each of the instances. If more than some threshold percentage of the instances inherit the slot, the slot is classified as a class slot, otherwise it is classified as an instance slot. The current threshold is 80%, and it has worked well in the sense that very few transmogrifications occur in practice (see Section 6.1 for more details).

The current implementation of the profiler is somewhat naïve but very effective. Except for a few slots, such as a *selected* slot, that the profiler knows Amulet might add to an instance, the profiler does not check to see if the instances add additional slots that the prototype does not contain. The profiler does maintain a list of common system-added slots so that these slots can be checked. The failure to check for additional slots could theoretically lead to a class being constructed with an insufficient number of slots. In turn, this oversight could lead to the transmogrification of all of the instances in the class.

However, as shown in Section 6.1, such transmogrifications have not been problematic in practice and our hypothesis stated at the beginning of this paper predicts that this strategy should not lead to many transmogrifications. Our hypothesis is that a graphical interface that contains a large number of objects will use a few prototype objects to stamp out thousands of objects with the same type. It is unlikely that an application will add or remove slots from these objects. Our empirical results reflect this.

In the future, we plan to make the profiler more sophisticated by having it make a preliminary pass through all the instances of a prototype in order to gather the complete set of slots used by all the instances. The profiler will then use this set of slots, rather than just the set of slots in the prototype, to perform its appraisal. This addition will handle the situation where a large number of instances all add an additional slot.

5.2 Selecting the Type of Object to Create

When an application creates an instance of an object, our scheme needs to determine whether the newly created object 1) will eventually serve as a prototype and create lightweight objects, 2) be a leaf in the prototype-instance hierarchy and be represented by a lightweight object, or 3) failing either of the above two conditions, be a heavyweight object. This choice is made by checking the name of the newly created object. Amulet gives a unique name to each object. The developer can specify a name, or, if one is not provided, Amulet assigns one automatically. When the profiler writes out a prototype's information, it also writes out the prototype's name. The code generator uses this name in the following fashion. The code generator first places a class declaration in a .h file and the class in a .cc file. The code generator then declares a dummy variable in the .cc file and initializes it by calling a function that stores in a global registry the class's name and a pointer to a static class method (Create) that returns an instance of the class.

When an object is created, the new name for it is specified. We check this new name. If it is in the registry, the newly created object is a lightweight-generating object, that is, an interior node in the inheritance hierarchy that has leaves immediately adjacent to it. We create a heavyweight object and give it a reference to the instance-generating function. Whenever the object's create method is called, it will create lightweight objects.

If the name does not appear in the registry, the system checks to see if the creating object is a lightweight-generating object. If it is, the new object is created as a lightweight. If it is not, a regular heavyweight object is created.

5.3 Initializing the Class Slots

The first time that a prototype class creates an instance of itself, it calls its static Initialize_Class_Slots methods. This method iterates through the class slots array. For each slot key in the array, it uses the class's *find_slot* method to locate the appropriate class slot. It then searches the prototype-instance hierarchy for the slot object corresponding to this key (since the slot is a class slot, it must be inherited from somewhere in the prototype-instance hierarchy). This search is performed by passing the slot key to the *find_slot* method of the heavyweight object that is associated with this prototype class (i.e., the heavyweight object that has a pointer to this class's Create method). The contents of located slot object are then copied into the class slot.

5.4 Initializing a Lightweight Object

Once created, the slots of a lightweight object must be initialized. This initialization is accomplished by copying the values of the slots in the prototype using the Initialize_Instance_Slots method. Recall that a lightweight class maintains an array of instance slot keys. The initialization procedure for a lightweight class simply iterates through this array. For each slot key, it retrieves the appropriate slot object from the prototype using the prototype's *find_slot* method. Similarly it retrieves the appropriate slot object from the instance using the lightweight object's *find_slot* method. It then copies the information from the prototype's slot object to the instance's slot object.

6. EMPIRICAL RESULTS

The feasibility of the lightweight scheme rests on two factors: 1) whether the number of transmogrifications is a relatively small percentage of the total number of objects, and 2) whether the performance degradation that results from the bridge object and from the transmogrifications is acceptable. The impact of using a bridge object is that operations affecting slots and triggers must go through another level of indirection. In other words, there is an additional virtual method call required to access slots and triggers (a virtual method call is required since the data object's type is unknown).

To help examine transmogrifications and performance, we took two existing Amulet applications, a finite state machine simulator and an interface builder called Gilt, and recompiled them using the class compilation scheme. Additionally, we created an "empty" application that simply initializes the Amulet run-time environment and then shuts down.

Each of the three applications was profiled using a script of actions (the script for the empty application was an empty script since the application simply starts up and shuts down). The applications were then recompiled with the classes created using the profile information. Finally the applications were rerun both with the same script and with a script of a different set of actions. Since the results could be made arbitrarily good by creating increasingly large numbers of application objects that would not be transmogrified, we created a relatively few number of application objects in each application, 21 in the finite state machine application and 100 in the Gilt application.

The applications were run on a Sun Sparc 5 with 32 megabytes of RAM.

6.1 Transmogrification Results

In each application we measured 1) the number of objects created, and 2) the number of objects transmogrified. The results are presented in the following table:

Application	**Number of Transmogrifications**	**Total Number of Objects Created**	**Percentage of Objects Transmogrified**
Empty Application	134	738	18%
Finite State Machine	162	929	17%
Interface Builder (Gilt)	359	4499	8%

Table 4: Transmogrification results for the sample applications.

The results show that a relatively small percentage of transmogrifications occurred in each application. In addition, almost all of the transmogrifications were in the parts of the Amulet prototype-instance hierarchy that are *not* used by the application. For example, Amulet provides a number of animation objects that are not used in any of the sample applications. Ordinarily these animation objects would be prototypes and hence would be interior nodes in the prototype-instance hierarchy. However, since no instances are created of these objects, they appear to the profiler to be instances. Hence the profiler creates lightweight objects for each of these prototype objects. Since Amulet goes ahead and customizes these objects, assuming that they will be prototypes, they end up being transmogrified. In the future we plan to examine ways that these transmogrifications could be avoided. However, the important point is that they do not constitute a large percentage of the objects. In addition, these transmogrifications represent a fixed cost that is incurred at start-up. They do not occur during the later execution of the interface.

Almost no transmogrifications occurred among the application objects during the user's interaction with the interface. This result is consistent with our belief that an application creates objects from a few prototypes and does not customize the objects by adding or deleting slots. It also supports our hypothesis that as the number of application objects is increased, the percentage of transmogrified objects will drop to almost 0.

6.2 Performance Results

We evaluated the performance of the class compilation scheme in two ways: 1) the user's subjective sense of the performance of the applications, and 2) the amount of time required to perform a single transmogrification. Although it is possible to perform a whole battery of benchmark tests on individual operations, we have found in the past that the best measure of performance is the user's subjective sense of performance. We found that there was no discernable difference between the original and the new "class" versions of these applications, either at start-up or during interactive operations. The performance of the class version is especially encouraging in light of the fact that it has not yet been extensively optimized. However, even with optimization, we doubt that there will still be any discernable difference in performance, since the performance of all applications was judged to be quite fast.

We did perform a number of benchmark timings between the original and the modified versions of Amulet and found a roughly 70% slowdown in performance for most operations. This finding is preliminary since the original version of Amulet is highly optimized, while the modified version of Amulet has only recently been completed and has not yet been optimized (for example, no attempt has been made to inline one and two line methods). The reason that interactive performance is not affected despite this slowdown is that most interactive operations affect only a small portion of a graphical interface, and in particular, a few objects. Consequently, the redisplay time dominates the time to actually alter the objects. This finding is confirmed by studies that have found that redisplay time accounts for roughly 70-80% of the time consumed by an operation [12,23].

The performance of the transmogrification routines on the transmogrification benchmark was also quite good. 10,000 instances of an object with 29 slots were created and transmogrified. The average time to transmogrify a single object was 1.22 milliseconds. Given these performance numbers and the number of objects transmogrified in the sample applications, it can be expected that transmogrification will add only a few fractions of a second to the overhead of an application. In addition, this overhead is for the most part incurred only at start-up.

7. CURRENT STATUS AND FUTURE WORK

This section summarizes our current progress and predicts the course of our future work.

7.1 Current Status

All of the mechanisms described in this paper are implemented. We have a working profiler, class generating program, and a modified version of Amulet that supports class objects and transmogrifications.

7.2 Future Work

The class compilation model described in this paper lays the foundation for making objects significantly more space efficient. Some of the optimizations we plan to pursue are:

Sharing trigger information. Most objects share the same set of triggers, the same set of default triggers, and the same trigger queue. Consequently, we plan to make these variables be class variables in lightweight objects.

Moving part management into the data objects. An Amulet object keeps track of parts in two ways, first in a part list and secondly in part slots that point to individual parts. In lightweight objects, these part slots can be organized into an array. Consequently, the part slot array can serve as the parts list and the explicit part list can be removed.

Using model dependencies for constraints to reduce constraint storage. A related paper describes how patterns can be used to generate constraint dependencies on demand rather than having to represent them explicitly in a constraint graph [11]. Not only do these "model" dependencies save storage directly, they will also enable some slots that currently have to be treated as instance variables to be treated as class variables instead. These slots are inherited slots that have constraint dependencies, and hence have to be allocated storage in the instance object in order to be able to store these explicit dependencies.

Making slots lightweight. Just as objects store trigger information that can be shared, slots also store trigger information that can be shared. We plan to extract this information and place it in either prototype slots or glyph objects, thus allowing the information to be shared. Similarly, some of the constraint information that is currently stored in the slots can, in most cases, be shared. We plan to use a transmogrification scheme similar to the one described in this paper to allow slots to be customized on demand.

8. CONCLUSION

This paper has described how type information that is obtained by profiling an application can be used to compile prototypes in a prototype-instance model into classes in a class-instance model. It also describes a transmogrification scheme that allows instances of these classes to be converted to a prototype-instance object if the instance must be customized in an unexpected way. This class compilation scheme lays the foundation for significantly reducing the amount of storage required by a prototype-instance model, since much of the type information that is currently stored in each individual object can instead be shared as class information. Once these storage optimizations are completed, the developer will be able to develop code using the prototype-instance model while still benefiting from the optimizations permitted the class-instance model

9. REFERENCES

1. Alan Borning. "The Programming Language Aspects of ThingLab; a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3, 4 (Oct. 1981), 353-387.
2. Alan Borning. "Classes versus Prototypes in Object-Oriented Languages." *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, (Nov.1986).
3. Paul R. Calder and Mark A. Linton. "Glyphs: Flyweight Objects for User Interfaces." ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'90, Snowbird, Utah, Oct., 1990, pp. 92-101.
4. Craig Chambers, David Ungar, and Elgin Lee. "An Efficient Implementation of SELF, A Dynamically-Typed Object-Oriented Language Based on Prototypes." *Sigplan Notices 24*, 10 (Oct. 1989), 49-70. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications; OOPSLA'89.
5. Craig Chambers and David Ungar. "Making Pure Object-Oriented Languages Practical." *Sigplan Notices 26*, 10 (Oct. 1991), 1-15. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications; OOPSLA'91.
6. Craig Chambers. "The Cecil Language: Specification & Rationale." Tech. Rept. 93-03-05, Computer Science Department, University of Washington, March, 1993.
7. Bjorn Freeman-Benson, John Maloney, and Alan Borning. "A Module Mechanism for Constraints in SmallTalk." *Sigplan Notices* 24, 9 (Oct. 1989). ACM Conference on Object-oriented Programming, Systems, Languages, and Applications; OOPSLA'89.
8. Bjorn Freeman-Benson. Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming. *OOPSLA/ECOOP'90 Conference Proceedings*, 1990, 77-88.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley, Reading, Massachusetts, 1995.
10. David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. "Profile-Guided Receiver Class Prediction." *Sigplan Notices 30*, 10 (Oct. 1995), 108-123. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications; OOPSLA'95.
11. Richard L. Halterman and Bradley T. Vander Zanden. "Using Model Dependency Graphs to Reduce the Storage Requirements of Dataflow Constraints in Prototype-Instance Systems". Submitted to OOPSLA'98.

12. Ralph Hill. "The Rendezvous Constraint Maintenance System." In ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'93, (Nov. 1993), Atlanta, GA, 225-234.
13. Urs Holzle and David Ungar. "Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback." *Sigplan Notices 29*, 6 (June 1994), 326-336. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation.
14. Urs Holzle and David Ungar. "A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance." *Sigplan Notices 29*, 10 (Oct. 1994), 229-243. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications; OOPSLA'94.
15. Scott E. Hudson. A System for Efficient and Flexible One-Way Constraint Evaluation in C++. Technical Report 93-15, Graphics Visualization and Usability Center, College of Computing, Georgia Institute of Technology, (April, 1993), 10 pages.
16. Scott Hudson and Ian Smith. "Ultra-lightweight Constraints." In *ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'96* (Oct. 1996), Seattle, WA.
17. Henry Lieberman. "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems." *Sigplan Notices 21*, 11 (Nov.1986), 214-223. ACM Conference on Object-Oriented Programming, Systems, and Applicationsl OOPSLA'86.
18. Brad Myers, Dario A. Guise, Roger Dannenberg, Bradley Vander Zanden, David Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces." *IEEE Computer 23*, 11 (Nov.1990), 71-85.
19. Brad Myers, Dario A. Guise, Andrew Mickish, and David Kosbie. "Making Structured Graphics and Constraints Practical for Large-Scale Applications." Tech. Rept. CMU-CS-94-150, Carnegie Mellon University Computer Science Department, May, 1994.
20. Brad A. Myers, Rich McDaniel, Robert Miller, Alan Ferrency, A. Faulring, Bruce Kyle, Andy Mickish, Alex Klimovitski, and P Doane. "The Amulet Environment: New Models for Effective User Interface Software Development". *IEEE Transactions on Software Engineering, 23, 6 (June 1997).*
21. David Ungar and Randall B. Smith. "SELF: The Power of Simplicity." *Sigplan Notices 22*, 12 (Dec. 1987), 227-241. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications; OOPSLA'87.

22. David Ungar, Randall B. Smith, Craig Chambers, and Urs Holzle. "Object, Message, and Performance: How They Coexist in Self." *IEEE Computer 25*, 10 (Oct. 1992), 53-64.

23. Brad Vander Zanden, Brad A. Myers, Dario Giuse and Pedro Szekely. "Integrating Pointer Variables into One-Way Constraint Models." *ACM Transactions on Computer Human Interaction*, 1, 2 (June 1994), 161-213.

24. David W. Wall. "Predicting Program Behavior Using Real or Estimated Profiles." *Sigplan Notices 22, 12* (Dec 1987), 227-241. ACM SIGPLAN'91 Conference on Programming Language Design and Implementation.