Alternative Object Organizations using Prototypes, Delegation and Split Objects

Hernán Astudillo R. John J. Shilling
hernan@cc.gatech.edu *shilling@cc.gatech.edu*

Technical Report GIT-CS-93/31
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

July 2, 1993

Abstract

Object-based (i.e. classless) models are very effective for elucidating requirements from users, and they support exploratory programming and rapid prototyping. On the other hand, class-based models are preferred to perform design and implementation, providing descriptive power and some types of error checking. We consider the evolution of object-based models into class-based production systems. One of the most difficult problems of this transition is the change from *explicit description of individuals* to *implicit description of class instances*. Reorganization support aims to make the system structures and properties *evident* and *enforceable*. Structural properties are useful to organize the code regardless of its meaning, and automated support can be enlisted to identify potential structures and properties, leaving the programmer with choice of alternative workspace organizations.

We analyze the organization and management of classless objects, regarding the goals of redundancy elimination and consistency maintenance, and how these goals are complicated by the existence of two mechanisms of object creation, cloning and extension (split objects). We present a classification of sharing and extension patterns in terms of the two basic mechanisms, and argue for a metrics-based approach to incremental reorganization. Finally, we propose in detail a set of abstractions with increasing descriptive power and consistency requirements: *groups* (untyped descriptions of structure and inheritance), *families* (partially typed descriptions of objects structure and inheritance, with monomorphic typing [1] and consistency maintenance), and *types* (fully typed descriptions of families interfaces, with polymorphic typing by subtyping and relating implementation hierarchy to typed interfaces).

[1] We call "monomorphic types" to types that do not allow subtypes, such as types in C (coercion is *not* subtyping)

1 Introduction

Recent research and practice have shown that object-oriented analysis (OOA) is a powerful tool for capturing system requirements in the early stages of development (for example, see [MP92, SM88, WBWW90, RBP+91] for some assessment of several OOA techniques). Some of these techniques (e.g. [Ner92, HH82]) make use of object-based models (i.e. with objects but no classes) due to their usefulness in representing real-world knowledge about the user's and system's domain and requirements. Object-based models allow the representation of concepts without a priori classifications; [Lie86]; support a customer dialog in which similarities to and differences from *concrete objects* can be detected and discussed; [BS84, Gom83]; and provide the flexible development needed for rapid prototyping [GS81].

Other experiences have shown [MP92, WBJ90, Boo90, Mey88] that development stages after requirements analysis instead benefit greatly of structural and behavioral formalization and predictability. *Classes* [GR89] turn the explicit description of individual objects into an intensional description of groups of objects; this simplifies the system description, allows code reuse, and supports change propagation. Typing [2] [Mey90a] (particularly static typing) is considered necessary for the development of large and commercial systems, since it supports reusing existing code without modifying it, allows to detect certain classes of errors by static examination, helps to document the system, and facilitates optimizations of the compiled code.

Without going as far as Meyer, who has argued [Mey90b] that classes and not objects should be the actual stuff of object-oriented development, we believe that formalization of structure and classification is necessary if a development effort is to result in an industrial-strength system. Short of imposing class-based approaches on early stages, an approach we want to avoid the only remaining option is to consider the evolution of object-based requirements models into class-based. Along this path, programming flexibility and exploratory attitudes are traded for automatic error checking, enforceable behavior restrictions, and efficiency. Our goal is to be able to develop systems in a relatively loose format, free of syntactic hassle, which could be systematically transformed into a more structured and robust form.

One of the most difficult problems of this transition is the change from *explicit description of individuals* to *implicit description of class instances*. Reorganization support aims to make the system structures and properties *evident* and *enforceable*. Structural properties are not enough, but they are useful to organizing the code regardless of its meaning. Automated support can be enlisted to identify potential structures and properties, but in the end it is the programmer who must give validity and meaning to the workspace organization.

In this paper, we consider the reorganization needs of an object-based system evolving into a type-based system. We start by analyzing the organization and management of isolated objects, regarding the issues of redundancy elimination, objects extension and consistency maintenance in workspaces of isolated objects. Finally, we propose in detail a set of abstractions with increasing descriptive power and consistency requirements: *groups*, untyped descriptions of structure and inheritance (supporting sharing, no consistency, and explicit object creation); *families*, partially typed descriptions of objects structure and inheritance, with monomorphic typing [3] and consistency maintenance (supporting sharing, consistency, and implicit object creation); and *types*, fully typed descriptions of families interfaces, with polymorphic typing by subtyping and relating implementation hierarchy to typed interfaces (defining conformance restrictions).

[2] In this paper we make a careful distinction between classes (an implementation mechanism based on encapsulation, templates and code extension) and typing (a relationship among protocol interfaces).

[3] We call "monomorphic types" to types that do not allow subtypes, such as types in C (coercion is *not* subtyping)

The remainder of this paper is organized as follows: Section 2 introduces the object model over which our mechanism is built; Section 3 examines the issues and mechanism involved in organizing and managing object workspaces; Section 4 describes our support for incremental organization, based on abstraction levels coupled with local-scope metrics; Section 5 describes *groups* and their operations and reorganization criteria; Section 6 discusses *families*, their representation, operations, reorganizations criteria, consistency maintenance, and their relationship with conventional classes; Section 7 introduces *types*, the relationship between types and conventional classes, their representation and operations, and their relationship with families; and 8 present our conclusions.

2 A simple object model

In this section, we define some terms that will be used throughout the paper. The basic entities are features and objects. A *features* is a named *reference* to an object. An *object* is either composite (a set of features) or a *method*. Features can be marked as "delegating features", and the object referenced by it is called a *parent object* of the feature holder.

The basic operations on an object are *cloning*, which creates a copy of it; addition and removal of features; modification of a feature, which alters its reference; and message sending. If the message receiver is a method, it is executed; if the receiver has a feature which matches the message, this one is forwarded to the object referenced by the feature; otherwise, it searches (recursively) among its parent objects for a matching feature to which forward the message.

A "cloned set" is created whenever an object is modified [4]. The set holds this initial object, plus any objects created by cloning a member of the set, minus any set members that are modified. Notice that our definition does not require the existence of a distinguished prototype, nor even the continued existence of the initial object.

3 Organization and management of objects

In object-based systems, the lack of (explicit) support for higher level abstraction and for intensional sets [5] creates two kinds of problems:

1. *Change propagation*: since there is no clear concept of equivalence or compulsory set membership ("clones set" are usually unknown to the programmer), any change to an object raises the question (or even worse, skips it altogether) if the change should also be applied to its clones or to its extensions (see Section 3.2).

2. *Replication control*: many objects may have *identical* immutable parts, taking up space due to its replication. The control, if not elimination, of replication, entails sharing some common elements, either explicitly through delegation or implicitly through a higher-level abstraction.

The severity of these two concerns depends greatly on the system policies for *sharing* among objects and for *objects extension*. In the following subsections, we introduce these two issues, and then discuss the effects of different combinations of sharing and extension policies.

[4] By adding, removing or modifying features.

[5] Sets created by description rather than enumeration.

3.1 Sharing

We separate the features of an object into *immutable* features, which are not meant to change during the object's lifetime and record the object's static properties, and *mutable* features, which do change and are used to record the object state. Clearly, these two kinds of features can be physically separated to improve space-use efficiency, since all immutable features are (by definition) identical among the members of a set of clones. [6]

Some mechanism to distinguish mutable and immutable features are feature annotations (indicating "const-ness") and system detection of common values among several objects; these mechanisms help to automate the identification and separation wherever possible.

Several approaches to this separation and sharing exist:

- *Ignored*: the issue is fully ignored. This alternative is mostly academic, since real systems would quickly suffer the effects of indiscriminate replication of essentially identical material.

- *System-simulated*: the problem is acknowledged, but is is considered just an implementation issue underlying a programmer's vision of objects as fully autonomous. This includes the Kevo language [Tai92], which provides sharing between apparently self-sufficient objects, and the "virtual copies" proposed by Mittal *et al* [MBK86].

- *Via delegation*: the sharing is explicit and visible to the programmer, who must determine what is shared and what is individual among clones. This is the model used in the Self language [US87], where "traits" objects hold shared slots, and "prototypes" and their clones hold individual state. The above mentioned approaches can be used to implement this one, as in fact Self [UCCH91] uses "maps" to support sharing among clones.

- *Implicit*: the issue of sharing is abstracted into membership in a group denoted by a common description of features. This is the case in class-based [CW85] languages like Smalltalk [GR89] and C++[Str86], where "classes" hold both shared members and an implicit prototype. Any of the above mentioned approaches can be used to implement this one, with the caveat that it be visible to the implementor but not to the programmer.

Figure 1 illustrates how a clones set (with members labeled A for simplicity) can be represented with either no-sharing, system-simulated sharing, or delegation-based sharing.

Figure 2 shows how the cloning and extension operations are combined: if a split object is to be cloned, then this operation must actually clone the hierarchy of data parents rather than just the object itself.

3.2 Objects extension

The basic operation to create new objects is *cloning*, which produces a copy another object. Systems with delegation also offer *extension*, where a new object is built as an incremental modification to another. (See Figure 3 for an example of creating a Student by either cloning an existing Student or extending an existing Person). In this case ("programming by extension"), new objects (or classes, if they exist) can be defined *as extensions of* existing ones; this relationship preserves the original object integrity, and allows automatic propagation of changes from an object's to its extensions.

[6] We would like to remark here that the distinction between mutable/immutable traits is orthogonal to the one between data/methods; for example, callbacks and error handlers are certainly mutable features even in conventional programming.

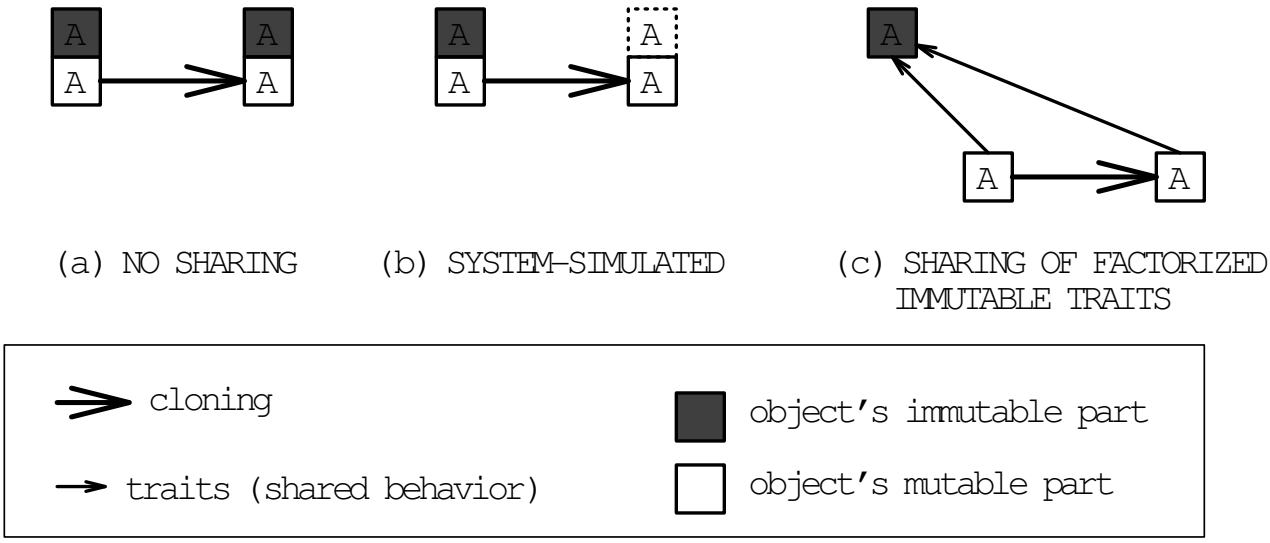

Figure 1: Factoring of immutable traits

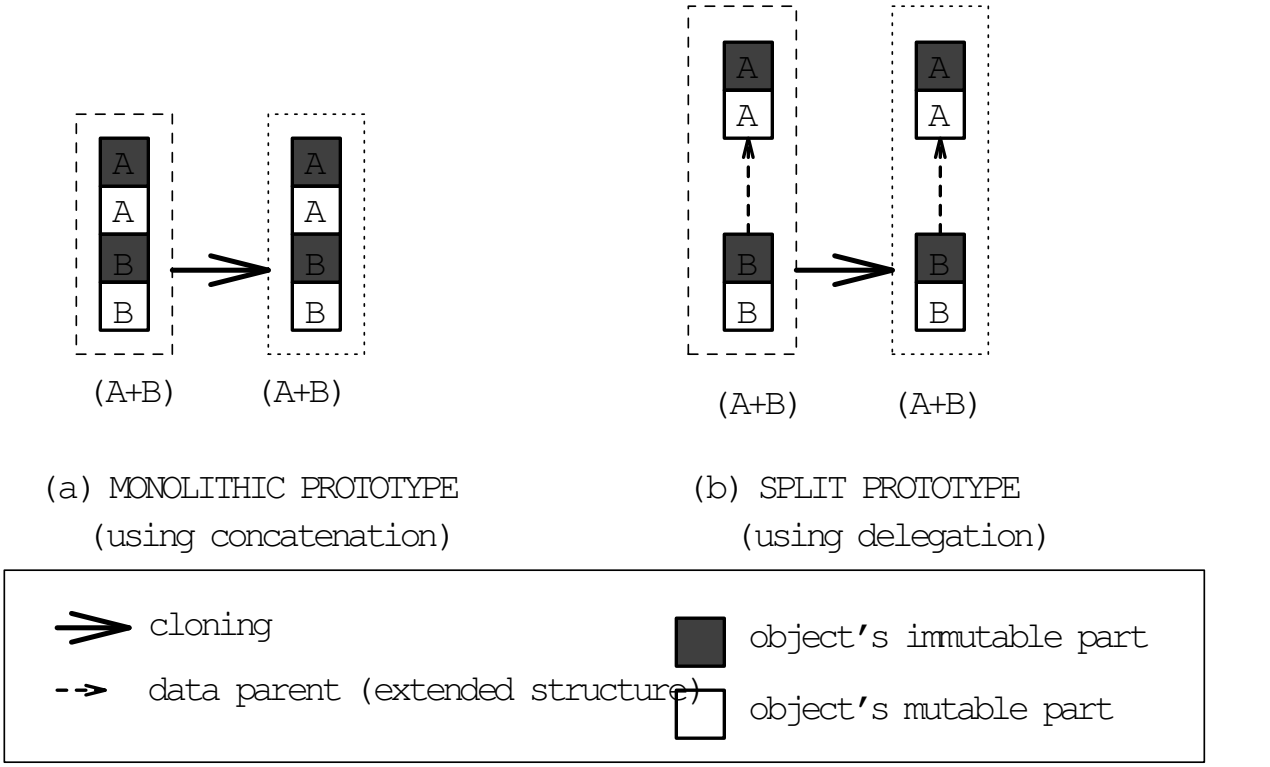

Figure 2: Cloning of extended objects

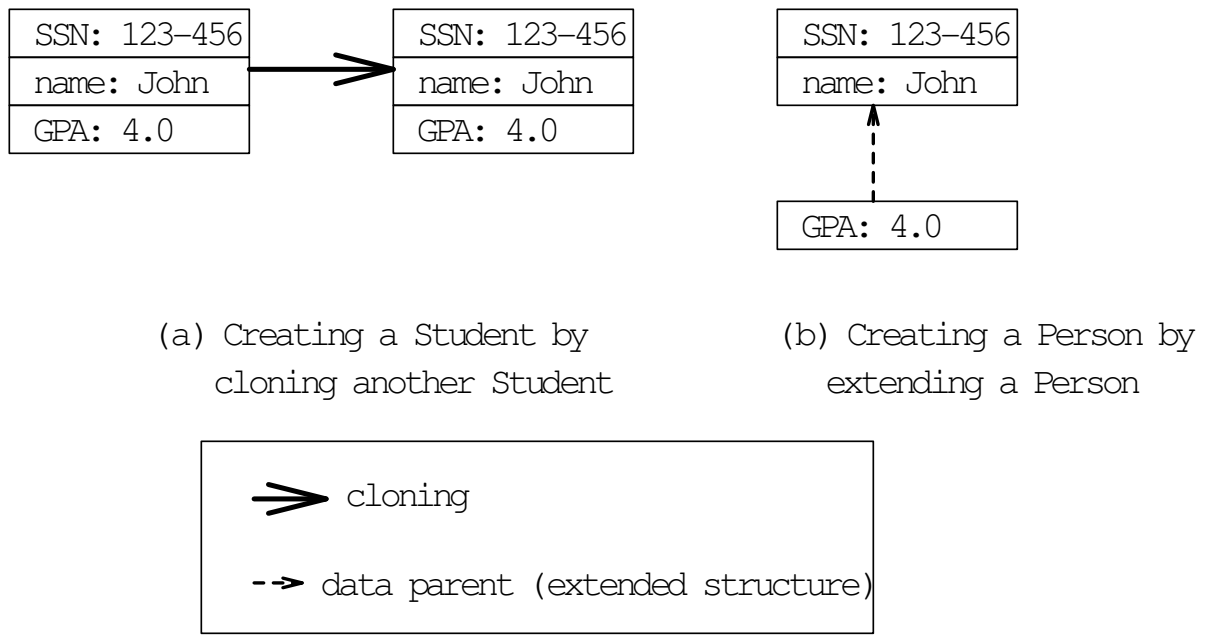

Figure 3: Cloning and extension

The extended object is composed of the desired object(s), plus a new object containing any additional features (see Figure 3). Several representation options exist:

- *Delegation*: an extended object is implemented as if having "components" [NGT92]: the new features are defined into an object, and the original object(s) become parents, remaining untouched. Dony *et al.* call these *split objects* [DMC92]. This avoids redundancy and facilitates change propagation, In Self, objects extended this way are called *data-parents* of their extensions.

- *Concatenation*: a new object is built by copying the object to be extended and then modifying it to satisfy the required extension. [7] This is the extension paradigm used in Kevo [Tai92]. This model is conceptual cleaner than delegation, since it supports self-sufficient objects by avoiding dependencies; on the other hand, it suffers of possible inconsistencies and certainly of replication. Notice that concatenation can be implemented using delegation and split objects; the key point is that the programmer can be safely unaware of this.

- *Implicit*: the process of extension itself is hidden by some abstraction. The programmer is not aware of object extension, but class extension can take its place. Smalltalk and C++ provide "subclassing", by which the implicit class-held prototypes are actually extended; instances created from a sub-class are themselves monolithic, even though classes themselves are both viewed and implemented as split objects [Tai92]. Implicit object extension can also be implemented using split objects or concatenation. [8]

3.3 Interaction of policies

Let's consider how the interaction of mechanism for redundancy elimination and for objects extension affects object organization modes. Our exposition is graphically shown in Figure 4,

[7] Not unlike the "cut-and-paste" paradigm.
[8] In fact, Stein has shown [Ste87] that inheritance is subsumed by delegation.

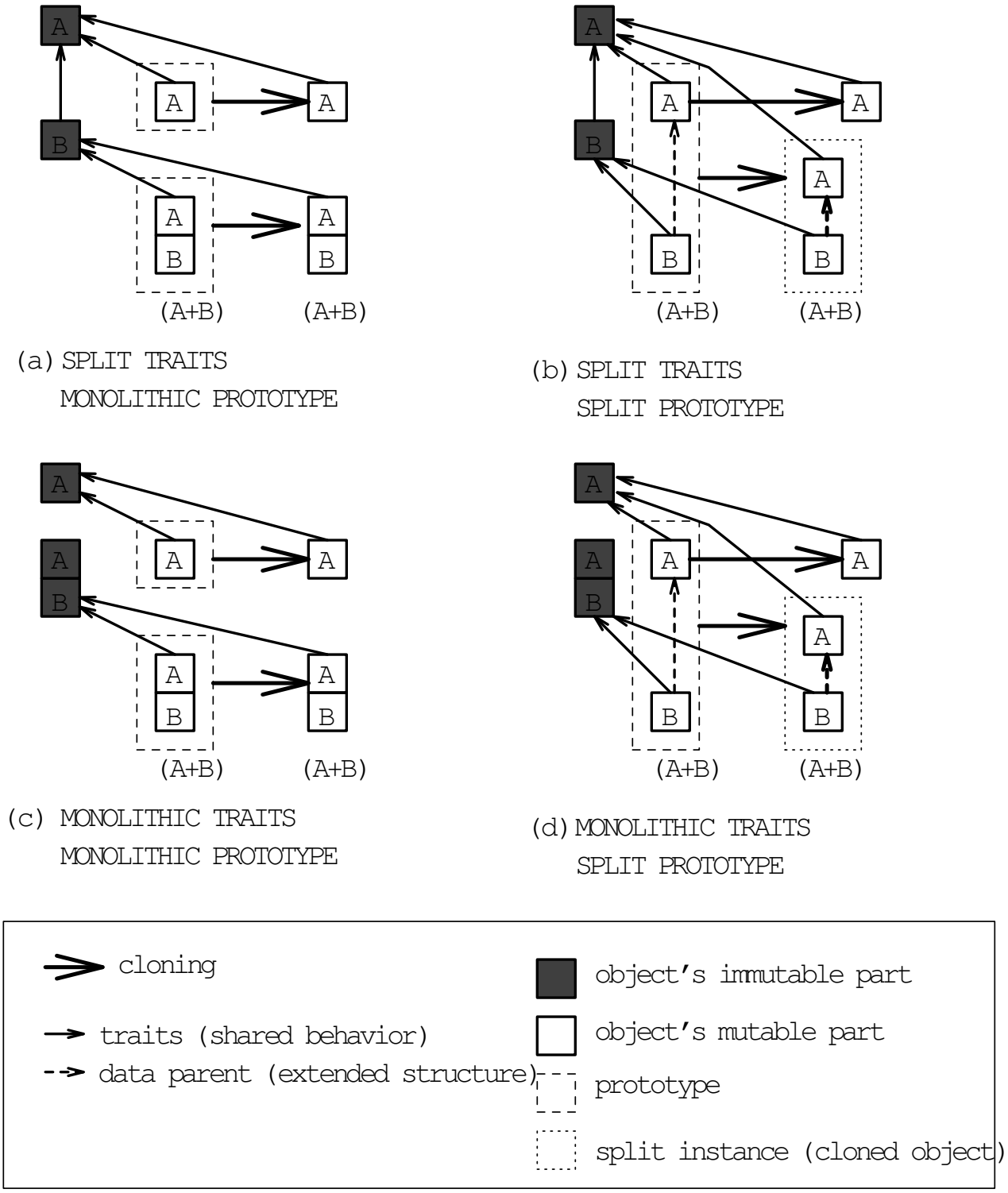

Figure 4: Extension and factorization

immutable part location		*monolithic prototype* (extension by concatenation)	*split prototype* (extension by delegation)
individual		Figure 2(a)	Figure 2(b)
shared (traits)	*monolithic*	Figure 4(c)	Figure 4(d)
	spli	Figure 4(a)	Figure 4(b)

Table 1: Combining extension and factorization

and it is summarized in Table 3.3. For all combinations of three factors, we evaluate the degree of change propagation and replication control available to the programmer. The factors are:

1. *Factorization*: whether the immutable features of an object reside in the object itself (the *individual*) or in a shared object (called *traits*, as in Self), which is accessed through delegating references;

2. *Traits splitting*: if the immutable features reside in a traits object, whether this object is a single object (a *monolithic traits* built using concatenation) or is a *split traits*, built using delegation.

3. *Prototype splitting*: whether the mutable features of an object reside in a single object (a *monolithic prototype*) or in several delegation-concatenated objects (*split prototype*).

The first row corresponds to self-sufficient objects, with no sharing. If only concatenation extension exists, then delegation (and sharing) have no place in the model; change propagation must be explicit, and replication is maximal. [9]If the model allows for split objects, then delegation is used only to support data-parents (i.e. objects sub-parts); propagation of changes from objects to their extensions is immediate, but changes to clones sets must be explicit.

The second row describes object models which allow sharing of a traits object with common immutable features, but the traits cannot be a split object itself. This means that a traits extending another one must replicate it; however, this will not produce as much replication as the previous cases. The propagation of changes from the originally extended traits must be done either separately (e.g. by a mechanism supporting a more abstract system on top of this), or in an ad hoc manner. The first one is the way that C++objects are *implemented* [Str86]: a large table referencing the class members, and instances with inherited members as sub-parts.

The third row considers object models which allow sharing of traits *and* that the traits itself be a split object. These two cases provide minimal replication for all immutable features in the object workspace. If objects are also allowed to be split, we have a situation of maximal change propagation for immutable features (which are propagated along the delegation lattice), minimal replication of features; propagation of instance structure changes requires the existence of higher levels of abstraction, such as classes. In fact, this last model is how C++programmers *perceive* their system: classes defined by extension, and instances with superclass parts as components.

Notice that case (d) in Figure 4 is at best redundant, and at worst ambiguous, since any instance of $(A + B)$ inherits actually two sets of $(A)_{traits}$: one from its own, and one from the $(A + B)_{traits}$. The solutions to this situation (see Figure 5) all involve making the (A) part of

[9]Unless the system itself implements one of the other models beneath the programmer's interface, like Self's "maps" [UCCH91].

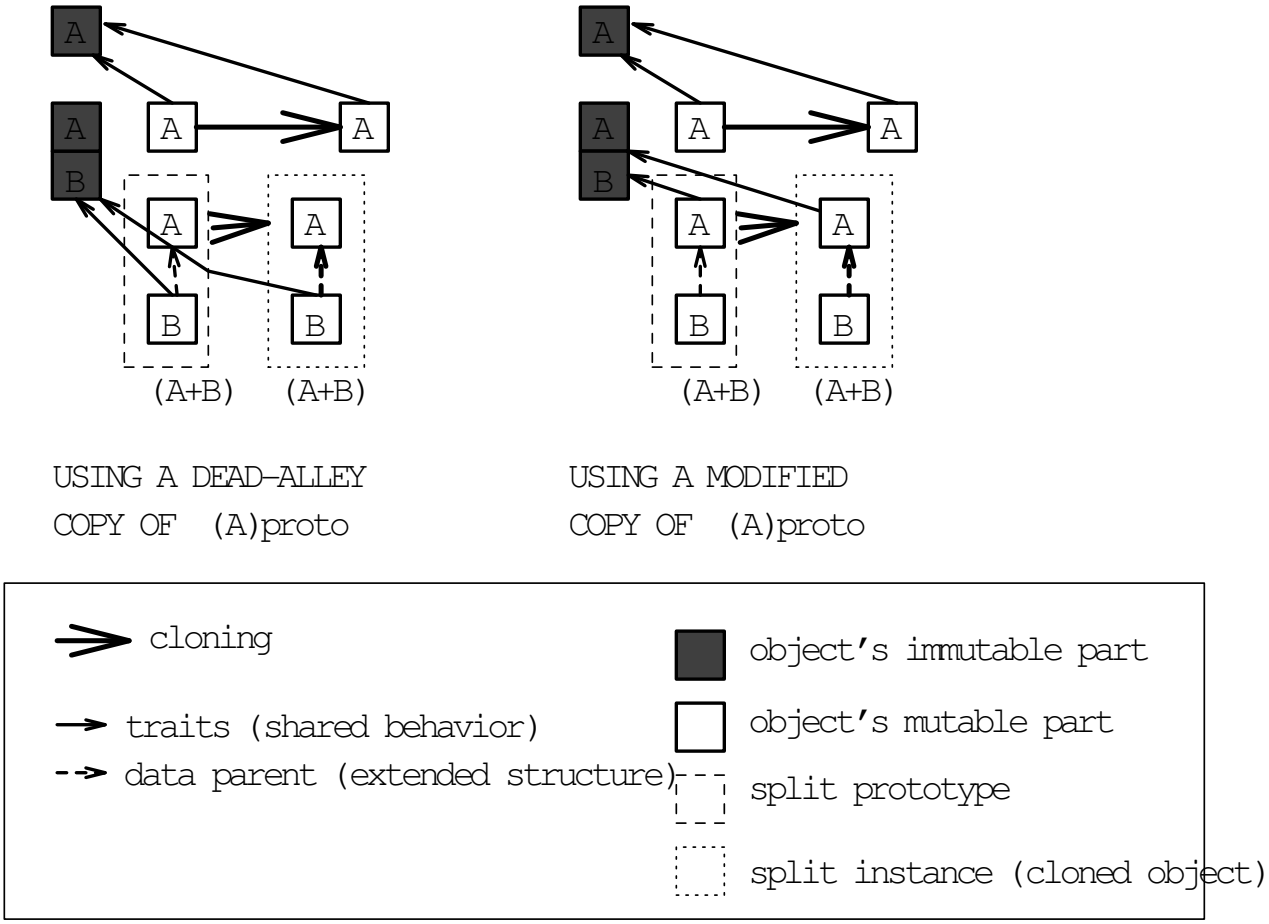

Figure 5: Solutions to Redundant Inheritance

$(A+B)_{proto}$ to be a *copy* of $(A)_{proto}$, rather than $(A)_{proto}$ itself; consequently, the advantage of split prototypes are lost. Notice that having a "blind-alley" copy of $(A)_{proto}$, i.e. with no parents, is valid because the "self" for $(A+B)$ instances refers to their (B) part.

These difficulties to combine split prototypes and monolithic traits seem to point a fundamental limitation to the split-ness of prototypes, namely that it cannot exceed their traits' split-ness.

4 Support for Increasing Organization

We propose a mechanism for reorganizing delegation-based workspaces into strongly typed systems has been presented. This evolution is based on three abstractions of increasing descriptive power and consistency requirements:

1. Untyped *groups* describing structure and inheritance.
2. Incrementally typed *families* describing objects structure and inheritance, which enable monomorphic typing.
3. Fully typed *types* describing family interfaces, which enable polymorphic typing by subtyping.

The consistency requirements of these three level increases with the available information: groups only require consistency of descriptions, families require consistency of instances with descriptions, and types require consistency of families with interfaces.

The distinction between inheritance for code reuse and for polymorphism is central to our effort. This reflects in the existence of monomorphic, code-oriented families versus polymorphic, interface-oriented types.

5 Basic Workspace Reorganization

A first level of support for classless programming can be achieved by providing operations that allow programmers to reorganize the object workspace to transform common aspects into shared ones.

Experience in classless programming [UCCH91] has found it reasonable to organize the workspace by providing two separate objects for each group of related and similar objects. The Self World [UCCH91] uses two elements for this purpose: a *traits* holding the shared behavior and data, and a *prototype* from which new objects can be cloned. Structural similarity is achieved among clones, and sharing is achieved by making all the clones to delegate on the traits.

This style of programming can be described as using class-like structures in a classless language. Unfortunately, once the moment for reorganizing the workspace arrives, the programmer must manipulate all the involved objects by himself, i.e. detect similarities, create the traits and prototype objects, and modify existing objects to inherit from the traits. Even worse, changes in the prototype do not reflect in existing clones, unless an excruciating search-and-modification operation is conducted.

We believe that reorganizations should not be postponed just because of their associated housekeeping chores. But the process cannot be fully automated either, because many decisions require an *understanding* of what some objects are intended for and how they will be used in the future. What is needed is a tool capable of *identifying* potential organizations, *creating* appropriate descriptions automatically, *administrating* these descriptions by providing operations on them, *maintaining* these descriptions consistently in the presence of changes, and *enforcing* them over all objects that are indicated as its instances.

We define a *group* as a collection of objects (its *instances*) with identical structure and common inheritance, plus a way of creating instances. A group is implemented by a *clonable* (used to create new instances) and a *sharable* (the common inheritance); the collection of instances is not explicitly handled. The reason for this is that groups are intended as low overhead, intermediate steps in further reorganizations.

Notice that the similarity of group instances with each other and with the group clonable holds only *when they are created*, since both structure and inheritance of instances may be modified later. In spite of this source of inconsistency, groups are the fundamental structure that can be discerned using structural comparisons, henceforth the primary concept to deal when reorganizing the workspace. In later sections, groups will be extended with consistency maintenance and domain restrictions.

In the following subsections we introduce a representation for groups, present some operations to create and maintain them, and explain how prospective groups can be identified from the workspace.

5.1 Representation of Groups

Groups are similar to the representation for classes in the Self World, as described in [UCCH91]. A group is defined using two objects, which can be directly examined and manipulated. They are:

- *Sharable*: an object which slots hold the data and methods shared by all the group instances. All the group instances inherit from this object.
- *Clonable*: an object which can be cloned to create new group instances. One of its slots is a delegating slot which references the group sharable; since it is copied when cloning, clones also delegate on the sharable.

Groups implementation has minimal overhead, because their definition does not involve any consistency rules to be represented and maintained.

5.2 Criteria for Similarity

The "similarity" of slots and objects is a concept that needs more clarification before the more complex operations and criteria are discussed. We define the following relationships among slots and objects.

1. Two slots are *name-similar* if they have equal name.
2. Two slots are *value-similar* if they have equal value.
3. Two slots are *similar* if they have equal name and value.
4. Two objects are *structure-similar* if their corresponding sets of *slot names* are equal (without considering ordering).
5. Two objects are *inheritance-similar* if their corresponding sets of *delegating slots* are equal, i.e. if their delegating slots are similar. This is the case when the two objects share inheritance.
6. Two objects are *similar* if they are structure-similar and inheritance-similar.

5.3 Operations on Groups

We provide several operations to create and manipulate groups. These operations are explained in greater detail in [ARS91].

1. *Factorization*: given a set of objects and a sharable, makes each object to inherit from the sharable and removes any "redundant" slots in each object (i.e. any method slot that is *value-similar* to a slot in the sharable).
2. *Completion*: given a set of objects and a clonable, each object is "completed" by adding to it all slots in the clonable.
3. *Synthesis*: given a set of objects, creates a new group from them. For this, all methods that are *value-similar* in all objects are factored out and inserted in a sharable, from which all (now) instances must inherit; and all other slots that are *name-similar* in all objects are copied in a new clonable.
4. *Distribution*: given a sharable, a clonable, a set of slots and a set of objects, removes those slots from the shared sharable and inserts them into each object that formerly inherited it from the sharable.

5.4 Identification of Potential Groups

To be able to identify groups and modify objects to become group instances, we can use the relationships previously defined among slots and objects. However, they only allow to identify "easy" groups (i.e. with very alike elements), so we extend the concept of similarity to handle approximate matchings. Again, further details may be found in [ARS91].

In this paper we present these relationship in informal terms:

1. If the slot set of O_1 is a subset of the slot set of O_2, then O_1 is *structure-smaller* than O_2.

2. If there are two objects O_1 and O_2, and neither of them is structurally smaller, but they do have common slots, then we say that O_1 is *X%-structure-similar* to O_2 where X is the percentage of the O_1 slot names that are in both objects. Note that this "similarity" is not symmetric.

Using these relationships, we can formulate several heuristics for identifying potential groups and performing the corresponding reorganizations. For example:

1. If several objects are *structure-similar*, then a group can be *synthesized* from them.

2. If several objects are *X%-structure-similar* both ways above certain threshold, then a group can be *synthesized* from them.

3. If an object O_1 is *structure-smaller* than an object O_2, then a *factorization* of O_1 by O_2 can be done (i.e. the "bigger" object becomes parent of the "smaller" one, which also loses its redundant slots).

6 Families

The groups described in Section 5 are useful to document design and provide a first level of reorganization based on structural similarity. An unfortunate characteristic of groups, however, is that their instance sets remain implicit. This allows for a low overhead, but also hampers any attempt at keeping consistency. Without the notion of instance set, the consistency of a group definition with its instances is an empty concept, since such instances are nowhere explicit. Modifications to the group's definition (e.g. to the clonable) do not reflect on the existing instances, but do affect new ones. To give meaning to consistency, we introduce the concept of family.

A *family* denotes both a set of objects (its *instances*) and a set of common properties on them, which must be kept throughout changes to workspace and to family definition. In particular, the properties described by a family are restrictions on the structure and on the slot values of its instances. It could be said that families are groups plus consistency plus simple typing.

Restrictions on instances structures are expressed by extending the concept of group (from the previous section) to enforce consistency, because families describe long-lived relations among definitions and instances, not just similarity at the creation of instances.

Restrictions on slots values are expressed using *tags*. Each slot S may be tagged with a reference to some family definition F; such a tag indicates that S can hold only references to instances of F.

The concept of family allows us to introduce the simplest kind of static type checking: monomorphic typing, where one object belongs to at most one type (no subtyping). We postpone

the discussion of polymorphism until we define types (in Section 7), because we want to separate inheritance for code reuse from inheritance for polymorphism.

There is no obligation for all slots to be tagged, so restrictions on domains can be done incrementally along with the progressive organization of the workspace. The classes found in conventional strongly-typed objected-oriented languages correspond to families with all slots already tagged.

Families can be created around an existing group, or they may be *synthesized* from a set of objects. These objects can themselves be family descriptions, providing a straightforward generation of sub- and super-families from existing ones.

In the following subsections we compare families with conventional classes, introduce a representation for families, present some operations to create and maintain them, explain how prospective families can be identified from the workspace. and consider the effect of normal workspace operations on the consistency of families.

6.1 Use of Families

Family tags are used to restrict the domain of slot values; in this respect they play a role similar to traditional type declarations. As in traditional type systems, the tag of a slot indicates a family whose slots may have tags which reference other families, and so forth. To put an end to this recursion, some primitive families are needed. Here we consider only one, INT, which "instances" are all the representable integer numbers.

The fact that families are tagged incrementally creates some problems when trying to compare tags using untagged slots. For conceptual uniformity, we assume that all objects which are not instances of some family are members of the ANY family.

Each object belongs to at most one family (or rather, to exactly one if we consider ANY). This saves the overhead of keeping track of every object, which number is presumably much bigger than the number of families.

Same as the slots in a family description, the argument and local objects of a method can be tagged. Once tags are used in a method, it becomes possible to perform some static type checking on it. Note that this static checking does not imply static binding, because (just like before the introduction of tags) the same messages are sent to the same objects: the only change is that slot lookup errors now can be detected before actually executing the method.

6.2 Extended Instances

Consider the following problem: given several objects, we create a new family whose slots are the minimum common denominator among all such objects. What if some of these objects end up having more slots than the new family?

To solve this problem, we must start from the fact that the programmer *explicitly* asked for these objects to be submitted to the family discipline, presumably because change control (e.g. renaming or retagging slots) is appropriate. Hence, the option of leaving the "bigger" objects out of the family is unacceptable. However, these objects do not fit the idea of instance as "valid example of the family properties".

To keep the family advantages for these objects while still enforcing the family structural properties on "normal" instances, we introduce the concept of *extended instances*. By doing so, we honor the programmer request while still making it possible to maintain consistency inside families.

We could go one step further and assume that extended instances are actually instances of subfamilies of the family under scrutiny (either existing subfamilies or subfamilies still undefined). However, this would be second-guessing the programmer, something that we are definitely avoiding to do. Instead, the operation to move instances into subfamilies can be used if necessary.

6.3 Representation of Families

Families are represented by five objects in the workspace, which can be browsed and (restrictedly) manipulated by the programmer. They are:

1. *Manager*: through which all the other family elements are manipulated.
2. *Sharable*: which holds the slots shared by all the instances.
3. *Clonable*: which can be cloned to create instances.
4. *Descriptor*: containing a set of pairs $< slot\ name, family\ tag >$, where the tag is a reference to a family definition. The family descriptor and clonable are *structure-similar*.
5. *Instance set (proper instance set)*: which references all the proper instances of the family.
6. *Extended instance set*: which references all the extended instances of the family.

The instance sets have a double purpose: facilitate consistency maintenance, and allow discretionary intensional manipulation (such as queries involving all instances of a family).

To enforce the family consistency while describing it with objects that reside in the workspace, the family definition should be accessed only through the family manager.

6.4 Criteria for Similarity

The addition of the tagging mechanism requires to redefine and extend the existing relationships:

1. Two slots are *tag-similar* if they have equal name and tag.
2. Two objects are *tag-similar* if their name-similar slots are also tag-similar.
3. An object is *tag-smaller* than another if its tagged slot set (names plus tags) is a subset of the second object's ($O_1 <_{tag} O_2$).
4. If there are two objects O_1 and O_2, and neither of them is tag-smaller, but they do have common slots with same tags, then we say that O_1 is *X%-tag-similar* to O_2 where X is the percentage of O_1 slots that are tag-similar in both objects. Note that this "similarity" is not symmetric.

6.5 Operations on Families and Instances

We provide several operations to create and manipulate families. These operations are explained in greater detail in [ARS91].

1. *Upgrading*: creates a new family around a group, i.e. the group is "upgraded" to a family. If the slots in the clonable and sharable reference objects which are instances of some families, then such families become the tags of those slots. The instance sets are initially empty.

2. *Synthesis*: creates a new family from a collection of objects, by synthesizing a group from them and then upgrading it to form the new family. The tag of the new slots can be deduced by examining their value in all instances; if there is some family to which all such values belong, it becomes the tag for that slot in the new family.

3. *Distribution*: this operation is like the distribution operation for groups, but the affected objects are (implicitly) the family instance sets. Also, it preserves the tags of all distributed slots.

4. *Classification*: [10] takes a set of objects and a threshold, and builds recursively a tree of families. First, a new family is synthesized from the given objects, and then all subsets of the set of extended instances that are *Y%-tag-similar* above the given threshold are submitted to the same classification operation. The final result of classification is a "tree" of families, each of them extending or redefining their respective superfamily slots.

Other operations for membership validation and set manipulation are also provided (see [ARS91]).

6.6 Operations among Families

1. *Factorization*: the sharable and description of each family F_i in F* are "factorized" to family F, by making each F_i to inherit from F and removing any redundant slots from their sharable.

2. *Distribution*: given a family, a set of subfamilies and a set of slots, copies these slots from the sharable of the family into the sharable of each subfamily. This operation allows later to specialize slots in subfamilies.

3. *Merge*: combines a collection of families into a single family whose description is the lowest common denominator of them, and put all instances "too large" for the in its set of extended instances.

6.7 Subfamilies

When a family is made a *subfamily* of another, two aspects must be taken care of: slot inheritance, and slot overriding.

The first consideration is whether only behavior is inherited, or both behavior and structure are inherited. The later option implies that some subfamilies should add more slots to their descriptions, and hence modify all of their instances. But let's consider for a moment the intention of the programmer: he is rearranging existing families, not starting from scratch; he perhaps synthesized these families from objects which were deemed already useful; adding slots to them would be at best useless. So, we opt for *inheriting only behavior* in subfamilies. This is one of the main reasons why families provide only monomorphic typing: instances of subfamilies are *not* guaranteed to have all the slots that instances of superfamilies have, but only those inherited from the superfamily sharable. (The state part could be totally different in principle.)

The second question is which slots to inherit. Since only behavior (and not structure) is inherited by the subfamily, the question can be answered by looking at the slots of the sharable of both families. A set difference will suffice, with a proviso: subfamily slots referencing methods

[10] Unfortunately, the word "familization" is too awkward for normal tastes.

can be eliminated only if they are *value-similar* to some inherited slot (because otherwise they are overriding it), but subfamily slots referencing data can be eliminated even if they are *tag-similar* (because data slots are overridden only if redefined with a different tag).

6.8 Identification of Potential Families

The heuristics used to identify groups can be extended to take into account the tags that family descriptors may have.

1. If several objects are *tag-similar*, then a family can be *synthesized* from them.
2. If several objects are *Y%-tag-similar* both ways above certain threshold, then a family can be *synthesized* from them.
3. At any time, a group can be *upgraded* to a family.
4. If there are several families whose sharable are *similar*, then they can be *merged*.
5. If there are several families whose sharable objects are *X%-structure-similar* both ways above certain threshold, then a new family can be synthesized from them.
6. If the sharable T_1 of a family is *structure-smaller* than the sharable T_2 of another family, then apply *factorization(slot set(*T_2*),*$\{T_1\}$*,*T_2*)*.

7 Types

Two of the most important concepts in object-oriented programming are inheritance and polymorphism [CW85]. Polymorphism can be achieved through a variety of means, such as overloading, genericity, subtyping, and simple lack of typing. Roughly speaking, we have polymorphism by subtyping if wherever an instance of a certain kind is expected, an instance of a subkind can be given instead.

Subtyping should represent two things: first, it needs that the interfaces of both types meet some criteria (like the one of "conformance" [Hor87]); and second, the use of a type in place of another must be *meaningful*. We stress the second point because the first one is usually the most (if not the only one) considered, whereas we believe that meaningfulness is the key to successful code organization. In general, inheritance for reuse of code is not enough to assert subtyping (see §7.1).

Families are useful for managing and documenting code implementation in terms of common structure, but they are not enough to describe subtyping unrelated to code inheritance. We introduce the concept of *type*, which functions are: specifying an interface, mapping this interface into an implementation, and specifying subtyping. These functions are realized by providing *type tags*, *type structures*, and *type mappings*.

Whereas a family represents a collection of objects whose consistency with a common declaration must be maintained, a type describes how a public interface specification (a description of behavior) is implemented by a family (a description of code). Types are defined with respect to families, not to single objects. An additional advantage of this is that such a type system probably would duplicate substantial parts of the family system.

Subtypes must be *explicitly* stated. Since subtyping requires conformance, a problem might arise when stating a subtyping relationship between types with incompatible interfaces. Luckily, the solution follows from the fact that subtyping must also be meaningful: if the programmer

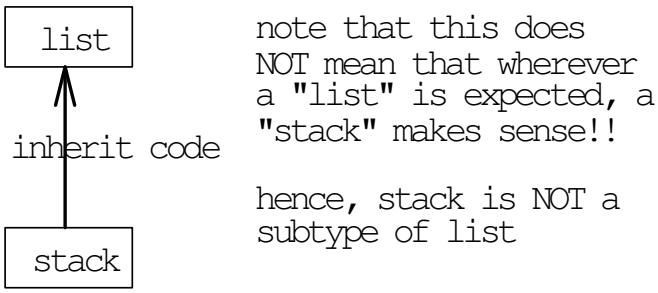

Figure 6: Code inheritance doesn't make subtype

knows that *sub* is a valid subtype of *super*, then he also knows how to "map" their interfaces. The eventual need for such mapping suggests how to implement types: as *mappings* between the slots of the type interface (the *type slots*) and the slots of a family (the *underlying family* of the type).

Types provide polymorphism via subtyping, i.e. a *type tag* T accepts instances of the underlying family of T *or of a family underlying any subtype of* T.

7.1 Types and Classes

Most conventional object-oriented languages [GR89] [Str86] [Mey88] combine "classes" (a code reuse abstraction) with "types" (a polymorphism abstraction) under a single linguistic construct. In these cases, each class implicitly defines a type, and subclassing (inheritance of implementation) implicitly defines subtyping. We believe that a cleaner separation of implementation and interface can be obtained from separating classes from types [CHC90] (like in POOL [AvdL90]).

We do not take issue with implicitly defining a type for each class; on the contrary, it seems very reasonable to assume that objects brought under common management are intended to have a similar use. However, it is clear from the examples above that subtyping cannot be assumed so easily, nor be deduced from structure or inheritance. Subtyping should ensure both conformance *and* meaningfulness. It is to fill these two requisites at a time that *families only define monomorphic* typing but *types define polymorphic typing*.

The following two examples show clearly that, in general, two families that are related by inheritance may not be valid subtypes, and two families that are structurally different maybe valid subtypes.

Figure 6 shows that a **stack** might be implemented using inheritance from **list**, but this does not imply that a **stack** can go wherever a **list** is expected. Conversely, Figure 7 shows that a **square** might be implemented without using code from **rectangle**, but it still holds that a **square** *can* go wherever a **rectangle** is expected (with appropriate member renaming, of course).

7.2 Representation of Types

A type is represented with the following objects:

1. *Manager*: through which all other definition elements are manipulated.

2. *Mapping*: a set of pairs $< name, reference >$ which indicate how type slots map to the underlying family. This allows to map types that are subtypes into families with different

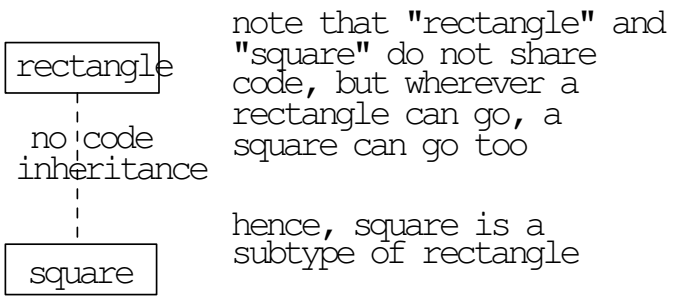

Figure 7: Subtypes don't need code inheritance

slot names altogether. Note that the second element of the pair references the objects or method referenced by the family's corresponding slot; if the family definition is changed, the appropriate slot in the type mapping must be changed too.

3. *Description*: a set of pairs $< name, tag >$, where the *tag* is a reference to a type definition. The tag of a slot S of a type must be a type whose underlying family is the same as the tag of the slot to which S maps. If more than one type maps to that family, then only the programmer can decide the proper type.

4. *Supertypes set*: a set of references to the definitions of all the type's supertypes.

7.3 Operations on Types

We provide several operations for creating and manipulating types.

1. *Create*: creates a new type using a family as its underlying family. The new type maps all the *public* slots of the family into similar slots of the type, and deduces their type tags from the family tags of the family.

2. *Create*: creates a type using a family as its underlying family, but using an explicit mapping instead of deducing it from the family,

3. *Subtype*: defines a type as subtype of another type.

7.4 Type Deduction from Mapping

To infer a type description for a type T from a type-to-family mapping over family F, we will consider each pair $< name_1, name_2 >$ in the type mapping, where $name_1$ is the type slot name and $name_2$ is the family slot name. These pairs are used as follows: if $name_1$ is the name of a valid slot in family F, then its tag tag_1 is looked: if the referenced family F_2 has one type T_2 that maps to it, then the pair $< name_1, name_2 >$ is entered in the type description of T. If F_2 is mapped by more that one type, then the programmer must select one.

7.5 Validity of Subtypes

A type can be subtype of more than one type (or of none at all). Unlike multiple inheritance in families, where the purpose was to reuse code from several families, "multiple supertyping" is used to indicate that a type's instances are valid instances for several other types, like shown in Figure 8.

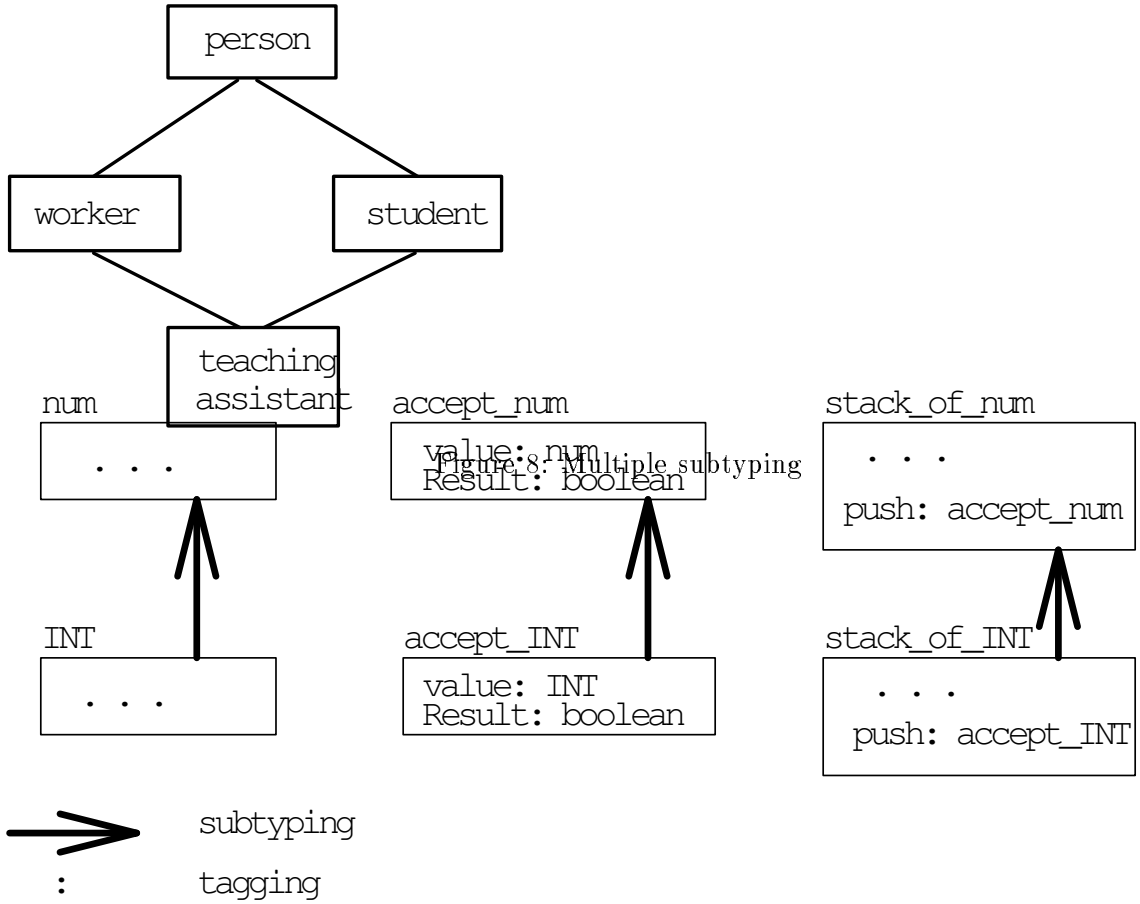

Figure 8: Multiple subtyping

Figure 9: Example of Covariance in Subtypes

The declaration of a subfamily by the programmer is enough to make it come true. However, the declaration of subtyping also requires that the involved types be checked for consistency.

To determine the validity of a subtyping declaration, we use the notion of *conformance* [Hor87], which we paraphrase as:

- T_1 conforms to T_2 iff the slot set of T_1 is a superset of the slot set of T_2, and for each slot S_i both in T_1 and T_2, the type tag of S_i in T_1 is a *subtype* of the type tag of S_i in T_2

Note that this definition requires at least one primitive type, so new types can be built from it. For simplicity, we just provide the type INT, which maps to the family INT.

The validity rule implies the *covariance rule* [Mey90a] for methods with the same name, because the type tag of a slot referencing a method P is the type tag of the method argument. In other words, if types *sub* is a subtype of *super*, and both types have a method P, then the argument type of P in *sub* must be a subtype of the argument type of P in *super*. Figure 9 shows a clear example.

We emphasize that covariance appears naturally from the definition of conformance, because a method's type is its argument's type.

8 Conclusions

In this paper, we have considered the reorganization needs of an object-based system evolving into a type-based system. We analyzed the organization and management of objects, regarding the

issues of redundancy elimination and change propagation in presence of sharing and extension. We proposed in detail a set of abstractions with increasing descriptive power and consistency requirements: *groups*, untyped descriptions of structure and inheritance (supporting sharing, no consistency, and explicit object creation); *families*, partially typed descriptions of objects structure and inheritance, with monomorphic typing and consistency maintenance (supporting sharing, consistency, and implicit object creation); and *types*, fully typed descriptions of families interfaces, with polymorphic typing by subtyping and relating implementation hierarchy to typed interfaces (defining conformance restrictions).

The distinction between inheritance for code reuse and for polymorphism is central to our effort. This reflects in the existence of monomorphic, code-oriented families versus polymorphic, interface-oriented types.

Finally, a major conclusion of our work is that *it is* possible to evolve free-format prototypes into strongly typed, production-quality systems. This should prove very beneficial for the use of rapid prototyping, because a common objection to it, namely its lack of safety unless fully rewritten, has been proved removable.

References

[ARS91] Hernán Astudillo R. and John J. Shilling. Criteria and operations for reorganization of classless systems. Technical Report GIT-CC-91/29, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA, Mar 1991.

[AvdL90] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In Norman Meyrowitz, editor, *OOPSLA/ECOOP '90 Conference on Object-Oriented Programming: Systems, Languages, and Applications/European Conference on Object-Oriented Programming*, pages 161–168. ACM SIGPLAN, ACM Press, Ottawa, Canada, Oct 21–25 1990.

[Boo90] Grady Booch. *Object-Oriented Design with Appications.* Benjamin/Cummings series in Ada and software engineering. Benjamin/Cummings, Redwood City, CA, 1990.

[BS84] Reinhard Budde and K. Sylla. From application domain modelling to target system. In Reinhard Budde, K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven, editors, *Approaches to Prototyping: Proceedings of the Working Conference on Prototyping*, pages 31–48, Berlin & New York, Namur, Belgium, 1983 1984. Gesellschaft fur Mathematik und Datenverarbeitung mbH, Springer-Verlag.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In POPL, editor, *17th ACM Symposium on Principles of Programming Languages.* ACM SIGACT and SIGPLAN, ACM Press, 1990.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dec 1985.

[DMC92] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages: From a new taxonomy to constructive proposals and their validation. In Andreas Paepcke, editor, *OOPSLA'92, 7th annual conference on object-oriented programming systems, languages, and applications*, pages 201–217. ACM SIGPLAN, ACM Press, Vancouver, British Columbia, Canada, 18–22 Oct 1992.

[Gom83] H. Gomaa. The impact of rapid prototyping on specifying user requirements. *Software Engineering Notes*, 8(2):17–28, 1983.

[GR89] Adele Goldberg and D. Robson. *Smalltalk-80: The Language.* Addison-Wesley series in computer science. Addison-Wesley, 1989.

[GS81] H. Gomaa and D. B. H. Scott. Prototyping as a tool in the specification of user requirements. In ICSE, editor, *5th International Conference of Software Engineering*, pages 333–342. IEEE and ACM SIGSOFT, IEEE Computer Society, 1981.

[HH82] J. W. Hooper and P. Hsia. Scenario-based prototyping of requirements identification. *Software Engineering Notes*, 7(5):88–93, 1982.

[Hor87] C. Horn. Conformance, genericity, inheritance and enhancement. In Jean Bezivin, editor, *European Conference on Object-Oriented Programming*, Lecture notes in

computer science ; 276., Berlin & New York, Paris, France, June 15-17 1987. Springer-Verlag.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In Norman Meyrowitz, editor, *OOPSLA '86 Conference Proceedings*, pages 214–223. ACM, ACM Press, Portland, Oregon, USA, Sept 29 – Oct 2 1986.

[MBK86] Sanjay Mittal, Daniel G. Bobrow, and Kenneth M. Kahn. Virtual copies: At the boundary between classes and instances. In Norman Meyrowitz, editor, *OOPSLA '86 Conference Proceedings*, pages 159–166. ACM, ACM Press, Portland, Oregon, USA, Sept 29 – Oct 2 1986.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Ciffs, NJ 07632, USA, 1988.

[Mey90a] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. Prentice Hall international series in computer science. Prentice Hall, Englewood Ciffs, NJ 07632, USA, 1990.

[Mey90b] Bertrand Meyer. Usenet article <454@eiffel.uucp>. Nov 1990.

[MP92] David E. Monarchi and Gretchen I. Puhr. A research typology for object-oriented analysis and design. *Communications of the ACM*, 35(9):35–47, Sep 1992.

[Ner92] Jean-Marc Nerson. Applying object-oriented analysis and design. *Communications of the ACM*, 35(9):63–74, Sep 1992.

[NGT92] Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, Sep 1992.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Ciffs, NJ 07632, USA, 1991.

[SM88] S. Shlaer and S. J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Prentice Hall, Englewood Ciffs, NJ 07632, USA, 1988.

[Ste87] Lynn Andrea Stein. Delegation is inheritance. In OOPSLA, editor, *OOPSLA '87: Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 138–146. ACM SIGPLAN, ACM Press, Orlando, FL, USA, 4–8 Oct 1987.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley series in computer science. Addison-Wesley, 1st edition, 1986.

[Tai92] Antero Taivalsaari. Kevo – a prototype-based object-oriented language based on concatenation and module operartions. Technical Report DCS–197–1R, University of Victoria,, Victoria, British Columbia, CANADA, June 1992.

[UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 4(3), Jun 1991.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In OOPSLA, editor, *OOPSLA '87: Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 227–241. ACM SIGPLAN, ACM Press, Orlando, FL, USA, 4–8 Oct 1987.

[WBJ90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):105–124, Sep 1990.

[WBWW90] Rebecca J. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, Englewood Ciffs, NJ 07632, USA, 1990.