# Systematic Change of Data Representation: Program Manipulations and a Case Study

William L. Scherlis *

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

**Abstract.** We present a set of semantics-based program manipulation techniques to assist in restructuring software encapsulation boundaries and making systematic changes to data representations. These techniques adapt abstraction structure and data representations without altering program functionality. The techniques are intended to be embodied in source-level analysis and manipulation tools used interactively by programmers, rather than in fully automatic tools and compilers.

The approach involves combining techniques for adapting and specializing encapsulated data types (classes) and for eliminating redundant operations that are distributed among multiple methods in a class (functions in a data type) with techniques for cloning classes to facilitate specialization and for moving computation across class boundaries. The combined set of techniques is intended to facilitate revision of structural design decisions such as the design of a class hierarchy or an internal component interface.

The paper introduces new techniques, provides soundness proofs, and gives details of case study involving production Java code.

## 1 Introduction

Semantics-based program manipulation techniques can be used to support the evolution of configurations of program components and associated internal interfaces. Revision of these abstraction boundaries is a principal challenge in software reengineering. While structural decisions usually must be made early in the software development process, the consequences of these decisions are not fully appreciated until later, when it is costly and risky to revise them. Program

* This material is based upon work supported by the National Science Foundation under Grant No. CCR-9504339 and by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0241. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, the National Science Foundation, or the U.S. Government. Author email: scherlis@cs.cmu.edu.

manipulation techniques can potentially support the evolution of internal interface and encapsulation structure, offering greater flexibility in the management of encapsulation structure.

This paper focuses on data representation optimizations that are enabled by structural manipulations. It presents three new results: First, it introduces two new program manipulation techniques, *idempotency* and *projection*. Second, it provides proofs for the two techniques and for a related third technique, *shift,* that was previously described without proof [S86,S94]. Finally, a scenario of software evolution involving production Java code is presented to illustrate how the techniques combine with related structural manipulations to accomplish representation change. The case study is an account of how the classes `java.lang.String`, for immutable strings, and `java.lang.StringBuffer`, for mutable strings, could have been derived through a series of structural manipulations starting with a single class implementing C-style strings. The intent of this research is to make the evolution of data representations a more systematic and, from a software engineering point of view, low-risk activity that can be supported by interactive source-level program analysis and manipulation tools. Source-level program manipulation presents challenges unlike those for automatic systems such as compilers and partial evaluators. The sections below introduce the program manipulation techniques, sketch their proofs, and give an account of the evolutionary case study.

## 2 Program Manipulation

Most modern programming languages support information-hiding encapsulations such as Java classes, Standard ML structures and functors, Ada packages, and Modula-3 modules. In this paper, we use a simplified form of encapsulation based on Java classes but with no inheritance (all classes are *final*), full static typing, and only private non-static fields (i.e., the data components, or *instance variables*, of abstract objects).

The internal representation of an abstract object thus corresponds to a record containing the instance variables. In Standard ML terminology, these classes correspond to `abstype`s or simplified `structure`s. We use the term *abstraction boundary* to refer informally to the interface between the class and its clients (the `signature` in Standard ML), including any associated abstract specifications. When the scope of access by clients to a class is bounded and its contents can be fully analyzed, then adjustment and specialization of the class *interface* (and its clients) becomes possible. For larger systems and libraries, this scope is effectively unbounded, however, and other techniques, such as cloning of classes (described below), may need to be used to artifically limit access scopes.

**Language.** We describe the manipulation techniques in a language-independent manner, though we adapt them for use on Java classes in the case study. We can think of an object being "unwrapped" as it passes into the internal scope of its controlling class, and "wrapped" as it passes out of the scope. It is convenient in this presentation to follow the example of the early Edin-

burgh ML and make this transit of the abstraction boundary explicit through functions `Abs` and `Rep`, which wrap and unwrap respectively. That is, `Rep` translates an encapsulated abstract object into an object of its data representation type, which in Java is a record containing the object's instance variables. For example, if an abstract `Pixel` consists of a 2-D coordinate pair together with some abstract intensity information, then `Rep` : `Pixel` $\rightarrow$ `Int` $*$ `Int` $*$ `Intensity` and `Abs` : `Int` $*$ `Int` $*$ `Intensity` $\rightarrow$ `Pixel`. Here "$*$" denotes product of types. The two key restrictions are that `Rep` and `Abs` (which are implicitly indexed on type names) can be called only within the internal scope of the controlling class and they are the only functions that can directly construct and deconstruct abstract values.

**Manipulations.** Manipulation of class boundaries and exploitation of specialized contexts of use are common operations at all stages of program development. We employ two general collections of meaning-preserving techniques to support this. The first collection, called *class* manipulations, is the focus of this paper. Class manipulations make systematic changes to data representations in classes. They can alter performance and other attributes but, with respect to overall class functionality (barring covert channels), the changes are invisible to clients. That is, class manipulations do not alter signatures (and external invariants), but they do change computation and representation.

The second class of manipulations are the boundary manipulations, which alter signatures, but do not change computation and representation. Boundary manipulations move computation into or out of classes, and they merge, clone, and split classes. The simplest boundary manipulations are *operation migration* manipulations, which move methods into and out of classes. Preconditions for these manipulation rules are mostly related to names, scopes, and types. A more interesting set of boundary manipulations are the *cloning manipulations*, which separate a class into distinct copies, without a need to fully partition the data space. Cloning manipulations introduce a type distinction and require analyses of types and use of object identity to assure that the client space can be partitioned—potentially with the introduction of explicit "conversion" points.

Boundary (and class) manipulations are used to juxtapose pertinent program elements, so they can be further manipulated as an aggregate, either mechanically or manually. Boundary manipulations are briefly introduced in [S94] and are not detailed here. It is worth noting, however, that many classical source-level transformation techniques [BD77] can be understood as techniques to juxtapose related program elements so simplifications can be made. Two familiar trivial examples illustrate: The derivation of a linear-space list reversal function from the quadratic form relies on transformation steps whose sole purpose is to juxtapose the two calls to list *append*, enabling them to be associated to the right, rather than the left association that is tacit in the recursion structure of the initial program. The derivation of the linear Fibonacci from exponential relies on juxtaposing two separate calls to $F(x-2)$ from separate recursive invocations. Various strategies for achieving this juxtaposition have been developed such as tupling, composition, and deforestation [P84,S80,W88].

**Class manipulations.** The *class* manipulations adjust representations and their associated invariants while preserving the overall meaning of a class. Details of these techniques and proofs are in the next two sections.

The *shift* transformation, in object-oriented terms, systematically moves computation among the method definitions within a class definition while preserving the abstract semantics presented to clients of a class. Shift operates by transferring a common computation fragment uniformly from *all* wrapping sites to *all* unwrapping sites (or vice versa). In the case study later in this paper, *shift* is used several times to make changes to the string representations. Shifts enable frequency reduction and localizing computation for later optimization. For example, in database design, the relative execution frequencies, use of space, and other considerations determine which computations are done when data is stored and which are done when data is retrieved. Shifts are also useful to accomplish representation change. For example, suppose a function `Magnitude` : `Int` * `Int` → `Real` calculates the distance from a point to the origin. If magnitudes are calculated when pixels are accessed, *shift* can be used to move these calls to wrapping sites. The representation of pixels thus changes from `Int` * `Int` to `Int` * `Int` * `Real`, with distances being calculated only when pixels are created or changed, not every time they are accessed.

The *project* transformation enables "lossy" specialization of representations, eliminating instance variables and associated computation. For example, in a specialized case where intensity information is calculated and maintained but not ultimately used by class clients, *project* could be used to simplify pixels from `Rep` : `Pixel` → `Int` * `Int` * `Intensity` to `Rep'` : `Pixel` → `Int` * `Int` and eliminate intensity-related calculations.

The *idempotency* transformation eliminates subcomputations that are idempotent and redundant *across* method definitions in a class. Idempotency is a surprisingly common property among program operations. Most integrity checks, cache building operations, and invariant maintaining operations such as tree balancing are idempotent.

## 3 Class manipulation techniques

Class manipulations exploit the information hiding associated with instance variables in a class (in Standard ML, variables of types whose data constructors are hidden in a **`structure`** or **`abstype`**). Significant changes can be made to the internal structure of a class, including introduction and elimination of instance variables, without effect on client-level semantics. Class manipulations are based on Hoare's venerable idea of relating abstraction and representation in data types.

**Shift.** The *shift* rule has two symmetrical variants. As noted above, the `Rep` and `Abs` functions are used to manage access to the data representation. A requirement on *all* instances of `Rep` or `Abs` (which are easy to find, since they are all within the syntactic scope of the class) thus has the effect of a universal requirement on all instances where an encapsulated data object is

constructed or selected. Let $T$ be the class (abstract) type of encapsulated objects and $V$ be the type of their representations. In the steps below, a portion of the internal computation of all methods that operate on $T$ objects is identified and abstracted into a function called *Span* that operates on $V$ objects. Note, in a Java class definition $V$ would be an aggregate of the instance variables (non-static fields) of the class. For our presentation purposes, we adapt the usual object-oriented notational convention to make the aggregate of instance variables an explicit value of type $V$. In the case study we make a similar adaptation for Java. This means we can require the fragment of shifted computation, abstracted as *Span*, can be a simple function, allowing a more natural functional style in the presentation of the technique. The effect of *Shift* is to move a common computation from all program points in a class where objects are unwrapped to program points where objects are wrapped (and vice-versa):

1. By other manipulations, local within each operation definition, establish that *all* sites of $\texttt{Rep} : T \to V$ appear in the context of a call to *Span*,

   $$\mathit{Span} \circ \texttt{Rep} : T \to V'$$

   where $\mathit{Span} : V \to V'$. That is, there is some common functional portion of computation of every method that occurs at object unwrapping. If this is not possible, the transformation cannot be carried out. Abstract all instances of this computation into calls to *Span*.
2. Replace **Abs** and **Rep** as follows. *All* instances of

   $$\texttt{Abs} : V \to T \quad \text{become} \quad \texttt{Abs}' \circ \mathit{Span} : V \to T'$$

   where $\texttt{Abs}' : V' \to T'$. *All* instances of **Rep** as

   $$\mathit{Span} \circ \texttt{Rep} : T \to V' \quad \text{become} \quad \texttt{Rep}' : T' \to V'$$

3. It is now established that *all* sites of $\texttt{Abs}' : V' \to T'$ are in the context of a call to *Span*,

   $$\texttt{Abs}' \circ \mathit{Span} : V \to T'$$

The *shift* rule, intuitively, advances computation from object unwrapping to object wrapping. The variant of *shift* is to reverse these three steps and the operations in them, thus delaying the *Span* computation from object wrapping to subsequent object unwrapping.

Although the universal requirement of the first step may seem difficult to achieve, but it can be accomplished trivially in those cases where *Span* is feasibly invertible by inserting $\mathit{Span}^{-1} \circ \mathit{Span}$ in specific method definitions. The *Span* operation is then relocated using the rule above. Later manipulations may then be able to unfold and simplify away the calls to $\mathit{Span}^{-1}$.

**Project.** The *project* manipulation rule eliminates code and fields in a class that has become dead, for example as a result of specialization of the class interface. The rule can be used when the "death" of code can be deduced only by analysis encompassing the entire class. Suppose the representation type $V$ can

be written as $V_1 * V_2$ for some $V_1$ and $V_2$. This could correspond to a partitioning of the instance variables (represented here as a composite type $V$) into two sets. If the conditions below are met, then *project* can be used to eliminate the $V_2$ portion of the representation type and associated computations. The rule is as follows:

1. Define **Rep** as **Rep** : $T \rightarrow V_1 * V_2$ and **Abs** as **Abs** : $V_1 * V_2 \rightarrow T$.
2. Represent the concrete computation of *each* method **op** that operates on any portion of the internal representation $V$ as

$$\mathbf{op} : V_1 * V_2 * U \rightarrow V_1 * V_2 * W$$

   where $U$ and $W$ are other types (representing non-encapsulated inputs and outputs). Then, by other manipulations, redefine **op** to calculate its result using two separate operations that calculate separate portions of the result,[1]

$$\mathbf{op}_1 : V_1 * U \rightarrow V_1 * W$$
$$\mathbf{op}_2 : V_1 * V_2 * U \rightarrow V_2$$

   Do this analogously for all operations **op** on internal representations $V_1 * V_2$. If this cannot be done for all such operations involving $V_1$ and $V_2$, then the rule cannot be applied.
3. Replace all operations **op** within the class by the new operations $\mathbf{op}_1$.
4. Simultaneously, replace all instances of **Abs** and **Rep** by new versions **Rep'** : $T \rightarrow V_1$ and **Abs'** : $V_1 \rightarrow T$.

The effect of *project* is to eliminate the $V_2$ portion of the overall object representation and all the computations associated with it. In Java, for example, $V_2$ would be a subset of the instance variables of a class. The *project* manipulation generally becomes a candidate for application after boundary manipulations have eliminated or separated all methods that depend on the particular instance variables corresponding to $V_2$. It is possible to think of *shift* and *project* as introduction and elimination rules for instance variables in the specific sense that, among its uses, *shift* can be used to introduce new fields and computations that could later be eliminated using *project*.

**Idempotency.** The *idempotency* rule is used to eliminate certain instances of idempotent computations such as integrity checks, caching, or invariant-maintaining operations. When an idempotent operation is executed multiple times on the same data object, even as it is passed among different methods, then all but one of the calls can be eliminated.

1. Define an idempotent *Span* function of type $V \rightarrow V$.

$$Span \circ Span = Span$$

2. Establish that each call of **Abs** appears in context **Abs** $\circ$ *Span or,* alternatively, establish that each call to **Rep** appears in context *Span* $\circ$ **Rep**. (These variants are analogous to the two variants of *shift.*)

[1] $\mathbf{op}(v_1, v_2, u) = [let\ (v_1', w) = \mathbf{op}_1(v_1, u)\ and\ v_2' = op_2(v_1, v_2, u)\ in\ (v_1', v_2', w)\ end]$

3. For one or more particular operation (method) definitions, establish in each that the call to *Span* can be commuted with the entire remaining portion of the definition.

$$\mathtt{op} = \mathit{Span} \circ \mathtt{op}' = \mathtt{op}' \circ \mathit{Span}$$

4. In each of the cases satisfying this commutativity requirement, replace op by op′. That is, replace the calls to *Span* by the identity function.

Examples: Performance-oriented calculations (spread over multiple operations) such as rebalancing search trees or rebuilding database indexes can be eliminated or delayed when their results are not needed for the immediate operation. An integrity check involving a proper subset of the instance variables can be eliminated from operations that do not employ those variables (i.e., delayed until a later operation that involves a variable in that subset). Note that this integrity check example depends on a liberal treatment of exceptions in the definition of program equivalence.

## 4 Correctness of Class Manipulations

The class manipulations alter data representations and method definitions in a class. Establishing correctness of the manipulations amounts to proving behavioral equivalence of the original and transformed class definitions. Since encapsulated objects are only unwrapped by methods within the class, the external behavior of a class can be construed entirely in terms of sequences of incoming and outgoing "exposed" values (to use Larch terminology) [G93]. This external class behavior is usually related to the internal definition of the class (i.e., the definitions of the instance variables and methods) through an explicit abstraction mapping, as originally suggested by Hoare [H72].

In the proofs below (as in the descriptions of the manipulations above), we employ a functional style in which methods are functional and objects are pure values. The set of instance variables of a Java class, which can be thought of as an implicit aggregate parameter and result, are made explicit in our rendering as new parameters and results of type $T$. This simple translation is possible (in our limited Java subset) because objects are pure values—i.e., there is no use of "object identity." Suppose, for example, that $T$ is the abstract type of a class, $U$ and $W$ are other types, and there are four methods in the class, which are functional and have the following signatures: $\mathtt{op}_1 : U \to T$, $\mathtt{op}_2 : T * T \to T$, $\mathtt{op}_3 : T \to U$, and $\mathtt{op}_4 : T \to T * W$. Thus, for example, $\mathtt{op}_1$ is a constructor and $\mathtt{op}_3$ does not alter or create $T$ objects.

Properties of objects in a class are expressed using an *invariant* $I(t, v)$ that relates abstract values $t$ (of type $T$ above) to representation (instance variables) $v$ of type $V$. Behavioral correctness *of a class* is defined in terms of preconditions on incoming exposed values, postconditions on outgoing exposed values, and a predicate $I$ that is an invariant and that relates abstract values with representations. If an abstract specification of the class is not needed, the proof can be obtained without reference to the abstract values $t$.

If $\mathbf{op}_i$ is a method, let $\overline{\mathbf{op}}_i$ be the implementation of the method, typed accordingly. For example, if $\mathbf{op}_i : U * T \to T * W$ then $\overline{\mathbf{op}}_i : U * V \to V * W$. Let $pre_i$ be a precondition on values of the exposed inputs to $\mathbf{op}_i$ and $post_i$ be a postcondition on the values of the exposed outputs from $\mathbf{op}_i$. For the invariant $I$ to hold for the class above ($\mathbf{op}_1$ through $\mathbf{op}_4$), the following four properties of $I$ must hold:

$$\begin{aligned} pre_1(u) &\Rightarrow I(\mathbf{op}_1(u), \overline{\mathbf{op}}_1(u)) \\ I(t_1, v_1) \wedge I(t_2, v_2) &\Rightarrow I(\mathbf{op}_2(t_1, t_2), \overline{\mathbf{op}}_2(v_1, v_2)) \\ I(t, v) &\Rightarrow post_3(\overline{\mathbf{op}}_3(v)) \\ I(t, v) &\Rightarrow I(\mathbf{op}_4(t), \overline{\mathbf{op}}_4(v)) \wedge post_4(\overline{\mathbf{op}}_4(v)) \end{aligned}$$

**Behavioral correctness.** Establishing behavioral correctness of a *manipulation rule* operating on a class definition amounts to showing that the relationships among $pre_i$ and $post_j$ values remain unchanged, even as the definition of $I$ changes in the course of transformation.

**Correctness of shift.** Let $I(t, v)$ be an invariant that holds for all methods in class $T$ prior to carrying out the transformation. The effect of *shift* is to create a new invariant $I'(t, v')$ that relates the new representation values $v'$ to the abstract values $t$. The proof proceeds by defining $I'$, showing it is invariant in the transformed class, and showing that it captures the same relationships between preconditions and postconditions as does $I$. We prove the first variant of *shift*; the proof for the second variant is essentially identical.

For a method $\mathbf{op} : T \to T$, for example, the *Span* precondition of the definition of the *shift* manipulation guarantees for some residual computation $r$ that

$$\begin{aligned} I(t, v) &\Rightarrow I(\mathbf{op}(t), \overline{\mathbf{op}}(v)) \\ &\Rightarrow I(\mathbf{op}(t), (r \circ Span)(v)) \end{aligned}$$

Define the post-manipulation invariant $I'(t, v') \equiv I(t, v) \wedge v' = Span(v)$. Now,

$$\begin{aligned} I'(t, v') &\equiv I(t, v) \wedge v' = Span(v) && \text{Definition of } I' \\ &\Rightarrow I(\mathbf{op}(t), (r \circ Span)(v)) \wedge v' = Span(v) && \text{Invariance property of } I \\ &\Rightarrow I(\mathbf{op}(t), r(v')) \\ &\Rightarrow I'(\mathbf{op}(t), (Span \circ r)(v')) && \text{Definition of } I' \\ &\Rightarrow I'(\mathbf{op}(t), \overline{\mathbf{op}}(v')) && \text{Definition of } \textit{shift} \end{aligned}$$

This establishes that $I'$ is invariant for the operation $\mathbf{op}$ in the class definition *after* it has been subject to the manipulation.

To understand the effect on postconditions, we consider the case of $\mathbf{op}_3$.

$$\begin{aligned} I(t, v) &\Rightarrow post_3(\overline{\mathbf{op}}_3(v)) && \text{Assumed} \\ I(t, v) &\Rightarrow post_3(r(Span(v)) && \text{Precondition of } \textit{shift} \\ I'(t, Span(v)) &\Rightarrow post_3(r(Span(v)) && \text{Definition of } I' \\ I'(t, Span(v)) &\Rightarrow post_3(\overline{\mathbf{op}}'_3(Span(v)) && \text{Definition of } I' \\ (\exists v)(v' = Span(v)) \wedge I'(t, v') &\Rightarrow post_3(\overline{\mathbf{op}}'_3(v') \end{aligned}$$

This is sufficient, since in the modified class definition, all outputs of methods are in the codomain of *Span*. These arguments are easily generalized to methods that include additional parameters and results.

**Correctness of idempotency.** There are two variants of the *idempotency* manipulation. We prove one, in which the second precondition (in the description of the technique) assures that for any $t$, $I(op(t), v) \Rightarrow (\exists v')(v = (Span \circ \hat{r})(v'))$ for some residual function $r$. Consider the case $\mathtt{op} : T \rightarrow T$. Because of the second precondition of the idempotency rule, op preserves the invariant,

$$\begin{aligned} I(t, v) &\Rightarrow I(\mathtt{op}(t), \overline{\mathtt{op}}(v)) \\ &\Rightarrow I(\mathtt{op}(t), (Span \circ \overline{\mathtt{op}}')(v)) \end{aligned}$$

For each method in which *Span* commutes (and in which *Span* can therefore be eliminated by the transformation), we need to show that the invariant is maintained, that is $I(t, v) \Rightarrow I(\mathtt{op}(t), \overline{\mathtt{op}}'(v))$. Assume inductively that invariant $I(t, v)$ holds for all class operations on type $T$ prior to carrying out the transformation. We proceed as follows:

| | |
|---|---|
| $I(t, v) \Rightarrow (\exists v')\ I(t, v) \ \wedge\ v = (Span \circ r)(v')$ | *idempotency* precondition |
| $\Rightarrow (\exists v')\ I(\mathtt{op}(t), (Span \circ \overline{\mathtt{op}}' \circ Span \circ r)(v'))$ | Invariant for op |
| $\Rightarrow (\exists v')\ I(\mathtt{op}(t), (\overline{\mathtt{op}}' \circ Span \circ Span \circ r)(v'))$ | Commutativity |
| $\Rightarrow (\exists v')\ I(\mathtt{op}(t), (\overline{\mathtt{op}}' \circ Span \circ r)(v'))$ | Idempotency of *Span* |
| $\Rightarrow I(\mathtt{op}(t), \overline{\mathtt{op}}'(v))$ | Definition of $v$ |

**Correctness of project.** The proof for the *project* manipulation is similar, and is omitted.

## 5 Strings in Java

We now present a modest-scale case study, which is a hypothetical re-creation of the evolution of early versions of the two Java classes used for strings, `String` and `StringBuffer`.[2] Java `Strings` are meant to be immutable and efficiently represented with minimal copying. For example, substring operations on `Strings` manipulate pointers and do not involve any copying. `StringBuffers`, on the other hand, are mutable and flexible, at a modest incremental performance cost. The typical use scenario is that strings are created and altered as `StringBuffers` and then searched and shared as `Strings`, with conversion operations between

[2] These two classes are part of an early version of the released Java Development Kit 1.0Beta from Sun Microsystems. Java is a trademark of Sun Microsystems. Since fragments of the code (string 1.51 and stringbuffer 1.21) are quoted in this paper, we include the following license text associated with the code: "Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved. Permission to use, copy, modify, and distribute this software and its documentation for non-commercial purposes and without fee is hereby granted provided that this copyright notice appears in all copies. Please refer to the file "copyright.html" for further important copyright and licensing information. Sun makes no representations or warranties about the suitability of the software, either express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, or non-infringement. Sun shall not be liable for any damages suffered by licensee as a result of using, modifying or distributing this software or its derivatives."

them. The data representations used in both classes facilitate this conversion by delaying or minimizing copying of structure. In particular, conversions from `StringBuffer` to `String` do not always result in copying of the character array data structure.

Both classes represent strings as character arrays called `value`. Both have an integer field called `count`, which is the length of the string in the array. The `String` class also maintains an additional integer field called `offset`, which is the initial character position in the array. Since `String`s are immutable, this enables substrings to be represented by creating new `String` objects that *share* the character array and modify `count` and `offset`.

The `StringBuffer` class, which is used for mutable strings, has three fields. In addition to `value` and `count`, there is a boolean `shared` that is used to help delay or avoid array copy operations when `StringBuffer` objects are converted into `String`s. The method used for this conversion does not copy `value`, but it does record the loss of exclusive access to the array `value` by setting the `shared` flag. If the flag is set, then a copy is done immediately prior to the next mutation, at which point the flag is reset. If no mutation is done, no copy is made. A simpler but less efficient scheme would always copy.

Selected steps of the evolutionary account are summarized below. The initial steps, omitted here, establish the initial string class, called `Str`, which manages mutable strings represented as null-terminated arrays, as in C. `Str` always copies character arrays passed to and from operations. (All boundary and class manipulation steps below have been manually carried out using our techniques, and one of these is examined more closely in the following section.)

**1.** Introduce `count`. In the initial class `Str`, trade space for time by using *shift* to replace uses of the null character by adding the private instance variable `count`. Simplifications following the shift eliminate all reference to and storage of null terminating characters.

**2.** Separate `Str` into `String` and `StringBuffer`. Label all `Str` variables in the scope in which `Str` is accessible as either `String` or `StringBuffer`, and introduce two methods to convert between them (both initially the identity function with a type cast). Since copy operations are done at every step, object identity cannot be relied upon, and this is a safe step. This process would normally be iterative, using type analysis and programmer-guided selection of conversion points. Those `Str` variables that are subject to mutating operations, such as `append`, `insert`, and `setCharAt` are typed as `StringBuffer`. Those `Str` variables that are subject to selection operations, such as `substring` and `regionMatches`, are typed as `String`. Common operations such as `length` and `charAt` are in both clones. Because copying is done at every step, conversion operations can always do an additional copy of mutable structures (i.e., `value`). This approach to allocating variables to the two clones enables use of boundary manipulations to remove mutating operations such as `setCharAt`, for example, from `String`. At this point, the two clones have different client variables, different sets of operations, but identical representations and representation invariants.

**3.** Introduce **offset**. Since **String**s are immutable, **substring** does not need to copy **value**. But elimination of copying requires changing the representation to include offsets as well as length. Use *shift* to accomplish the representation change preparatory to eliminating the copying. (This is the step that is detailed below.)
**4.** Introduce **shared**. In **StringBuffer**, **value** is copied prior to passing it to the **String** constructor. This is necessary because **value** would otherwise be aliased and subject to subsequent mutation, which would violate the mutability invariant of **String**. In fact, the copying needs to be done only if **value** is subsequently mutated, and could be delayed until then. Use *shift* to relocate the copying to whatever **StringBuffer** method immediately follows the conversion, adding the **shared** flag. This delays the copy by "one step." Then use *idempotency* to delay that copying "multiple steps" (by commuting it within all calls to non-mutating access methods) until a **StringBuffer** operation that actually does mutation.
**5.** Add specialized methods. Identify conversion functions from various other common types to **char[]**. For both string classes, use boundary shifts to compose these conversion functions with the constructors and with commonly used functions such as **append** and **insert**. Then use additional boundary shifts to relocate these composed functions as public methods, naming them appropriately (mostly through overloading). Then, unfold (inline) the type conversion calls contained in these methods and specialize accordingly. (This introduces many of the large number of methods defined in the two classes.)
**6** (hypothetical). Eliminate **value**. Suppose that in some context only the lengths of **StringBuffer**s were needed, but not their contents. That is, after a series of calls to methods such as **insert**, **append**, **setCharAt**, **reverse**, and **setLength**, the only accessor method called is **length**. *Clone* and *project* can be used to create a specialized class for this case.

## 6 A Closer Look at Offsets in Class String

We now consider more closely step 3 above, which introduces the **offset** instance variable as an alternative to copying when substrings are calculated. This step must include, for example, modification of all character access operations to index from **offset**. This account illustrates how *shift* can be employed to introduce offsets into the string representation. An initial Java version of **substring** in class **String** could look like this:

```
public String substring (int b, int e) {
   // b is first char position; e-1 is last character position.
   if ((b < 0) || (e > count) || (b > e))
        {throw new IndexOutOfBoundsException (endi); }
   int ncount = e - b;
   char res[] = new char[ncount];
   ArrayCopy (value, b, res, 0, ncount);
   return new String (res, ncount);
 }
```

`ArrayCopy` copies `ncount` characters from `value` to `res` starting at offsets `b` and 0 respectively.

To facilitate presentation consistent with the manipulation rules, the code below explicitly mentions in special brackets those `(- instance variables -)` that are referenced or changed. This allows a more functional treatment of methods. We also usually abbreviate `{- integrity checks -}`. Thus the code above could be rendered more succinctly as:

```
public String substring (int b, int e) (- value, count -) {
  {- Check (b, e, count) -}
  int ncount = e - b;
  char res[] = new char[ncount];
  ArrayCopy (value, b, res, 0, ncount);
  return new String (res, ncount);        }
```

The principal steps for *shift*, recall, are (1) defining a suitable *Span* function, (2) adapting code so it appears in all the required places, and (3) carrying out the *shift* transformation by relocating the *Span* calls. We then (4) simplify results.

**(1) Define Span.** The motivating observation is that the array copying is unnecessary when a substring is extracted. We therefore abstract the instances of array copy code into a new method definition and use that as *Span*. The original code sites will be replaced by calls.

```
private (char[],int) Span (- char[] nvalue, int ncount, int noffset -) {
  char res[] = new char[ncount];
  ArrayCopy (nvalue, noffset, res, 0, ncount);
  return (res,ncount);    }
```

(An additional notational liberty we take is to allow functions to return a *tuple* of results. For this variant of *shift*, the parameters of *Span* are exactly the new set of instance variables, and the results are the old instance variables.

**(2) Call Span.** The method abstraction step needs to be done in a way that introduces *Span* calls into all methods that construct or mutate string values (i.e., alter `value` or `count`). Note that `substring` does *not* mutate or construct string values directly; it uses a `String` constructor:

```
public String (nvalue, ncount) (- value, count -)  {
  value = new char[ncount];
  count = ncount;
  ArrayCopy (nvalue, 0, value, 0 count);     }
```

After *Span* is abstracted:

```
public String (nvalue, ncount) (- value, count -)  {
  (value, count) = Span (nvalue, ncount, 0);   }
```

When *Span* has an easily computed inverse function, then *Span* can always be introduced trivially into otherwise less tractable cases as part of a composition with the inverse. But this will likely necessitate later code simplification.

**(3) Do the Shift.** Once all the *Span* calls have been inserted, doing the *shift* is a mechanical operation. The effect is as follows:

- The class now has three private instance variables, **nvalue**, **ncount**, and **noffset**, corresponding to the parameters of *Span*.
- All *Span* calls are replaced by (simultaneous) assignments to the new instance variables of the actual parameters.
- In each method that accesses instance variables, the first operation becomes a call to *Span* that computes **value** and **count** (now ordinary local variables) in terms of the instance variables.

For example, here is the transformed definition of **substring**:

```
public String substring (int b, int e) (- nvalue, ncount, noffset -) {
  (char value[], int count) = Span (nvalue, ncount, noffset);
  {- Check (b, e, count) -}
  int c = e - b;
  char res[] = new char[c];
  ArrayCopy (value, b, res, 0, c);
  return new String (res, c);       }
```

**(4) Simplify.** Unfolding *Span* and simplifying:

```
public String substring (int b, int e) (- nvalue, ncount, noffset -) {
  {- Check (b, e, ncount) -}
  int c = e - b;
  char res[] = new char[c];
  ArrayCopy (nvalue, noffset, res, 0, c);
  ArrayCopy (res, b, res, 0, c);
  return new String (res, c);       }
```

This motivates our introducing a new string constructor that can take an offset, **public String(nvalue,ncount,noffset)**. This is done by abstracting the last two lines of **substring** and exploiting the idempotency of array copying. After simplifications:

```
public String substring (int b, int e) (- nvalue, ncount, noffset -) {
  {- Check (b, e, ncount) -}
  return new String (nvalue, e - b, b + noffset); }
```

In the case of the **charAt** method in **String**, we start with:

```
public char charAt(int i) (- value, count -) {
  {- Check (i, count) -}
  return value[i];       }
```

This does not create or modify the instance variables, so no call to *Span* is needed before the shift. It does, however, access instance variables, so *shift* inserts an initial call to *Span*:

```
public char charAt(int i) (- nvalue, ncount, noffset -) {
  (char value[], int count) = Span (nvalue, ncount, noffset);
  {- Check (i, count) -}
  return value[i];       }
```

Simplification entails unfolding *Span*, eliminating the array copy operation, and simplifying:

```
public char charAt(int i) (- nvalue, ncount, noffset -) {
  {- Check (i, ncount) -}
  return nvalue[i + noffset];       }
```

## 7 Background and Conclusion

**Background.** There is significant previous work on synthesizing and reasoning about encapsulated abstract data types. Class manipulations are most obviously influenced by Hoare's early work [H72]. Most related work focuses either on synthesizing type implementations from algebraic specifications or on refinement of data representations into more concrete forms [D80,MG90,W93]. But evolution does not always involve synthesis or refinement, and the techniques introduced here are meant to alter (and not necessarily refine) both structure and (internal) meaning.

Ideas for boundary manipulations appear in Burstall and Goguen's work on theory operations [BG77]. Wile developed some of these ideas into informally-described manipulations on types that he used to provide an account of the heapsort algorithm [W81]. He used a system of manipulating flag bits to record whether values have been computed yet in order to achieve the effect of a primitive class manipulation. For boundary manipulations, we build on several recent efforts which have developed and applied concepts analogous to the *operation migration* techniques [GN93,JF88,O92].

Early versions of the boundary manipulation and *shift* techniques were defined (without proof) and applied to introduce destructive operations in programs [JS87,S86] and in a larger case study in which a complex data structure for the text buffer of an interactive text editor was developed [NLS90].

The program manipulation techniques we introduce for classes and other encapsulations are meant to complement other manipulation and specialization techniques for achieving local restructuring and optimizations that do not involve manipulating encapsulations.

**Conclusion.** The techniques described in this paper are source-level techniques meant to be embodied in interactive tools used with programmer guidance to develop and evolve software. The intent is to achieve a more flexible and exploratory approach to the design, configuration, implementation, and adaptation of encapsulated abstractions. Manipulation techniques such as these can decrease the coupling between a programmer decision to perform a particular computation and the related decision of where the computation should be placed with respect to class and internal interface structure.

**Acknowledgements.** Thanks to John Boyland and Edwin Chan, who contributed valuable insights. Thanks also to the anonymous referees for helpful comments.

## References


[BG94] R.W.Bowdidge and W.G. Griswold, Automated Support for Encapsulating Abstract Data Types. ACM SIGSOFT Symposium Foundations of Software Engineering, 1994.

[BD77] R.M. Burstall and J. Darlington, A transformation system for developing recursive programs. JACM 24, pp.44-67, 1977.

[BG77] R.M. Burstall and J. Goguen, Putting theories together to make specifications. IJCAI, 1977.

[D80] J. Darlington, The synthesis of implementations for abstract data types. Imperial College Report, 1980.

[GN93] W.G. Griswold and D. Notkin, Automated assistance for program restructuring. ACM TOSEM 2:3, 1993.

[G93] J.V. Guttag and J. J. Horning, et al., Larch: Languages and Tools for Formal Specification, Springer-Verlag, 1993.

[H72] C.A.R. Hoare, Proof of correctness of data representations. Acta Informatica 1, pp. 271-281, 1972.

[JF88] R. Johnson and B. Foote, Designing reusable classes. Journal of Object-Oriented Programming, June/July 1988.

[JS87] U. Jørring and W. Scherlis, Deriving and using destructive data types. Program Specification and Transformation, Elsevier, 1987.

[K96] G. Kiczales, Beyond the Black Box: Open Implementation, IEEE Software, January 1996.

[MG90] Carroll Morgan and P.H.B. Gardiner, Data refinement by calculation. Acta Informatica 27 (1990).

[MG95] J.D. Morgenthaler and W.G. Griswold, Program analysis for practical program restructuring. ICSE-17 Workshop on Program Transformation for Software Evolution, 1995.

[NLS90] R.L. Nord, P. Lee, and W. Scherlis, Formal manipulation of modular software systems. ACM/SIGSOFT Formal methods in software development, 1990.

[O92] W. Opdyke, Refactoring Object-Oriented Frameworks. PhD Thesis, University of Illinois, 1992.

[P86] R. Paige, Programming with invariants. IEEE Software 3:1, 1986.

[P84] A. Pettorossi, A powerful stragey for deriving efficient programs by transformation. ACM Symposium on Lisp and Functional Programming, 1984.

[S80] W. Scherlis, Expression procedures and program derivation. Stanford University technical report, 1980.

[S86] W. Scherlis, Abstract data types, specialization, and program reuse. Advanced programming environments, Springer, 1986.

[S94] W. Scherlis, Boundary and path manipulations on abstract data types. IFIP 94, North-Holland, 1994.

[W88] P. Wadler, Deforestation: Transforming programs to eliminate trees. European Symposium on Programming, Springer, 1988.

[W81] D.S. Wile, Type transformations. IEEE TSE SE-7, pp.32-39, 1981.

[W93] K.R. Wood, A practical approach to software engineering using Z and the refinement calculus. ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1993.