A Language Supporting Direct Manipulation of Component-based Object Construction and Deconstruction in Collaborative Croquet Spaces

Howard Stearns, Joshua Gargus, Martin Schuetze
Division of Information Technology
University of Wisconsin-Madison
[hstearns, gargus]@wisc.edu, martin.schuetze@gmail.com
and
Julian Lombardi
Office of Information Technology
Duke University
julian@duke.edu

Abstract

We describe the language architecture of Brie, a framework for authoring 3D spaces and user interfaces. Brie is designed to take advantage of the unique social characteristics of the Croquet collaboration architecture, in particular by lowering the bar for content creation, thus greatly expanding the community of content developers. This is achieved through a 3D direct-manipulation interface to objects. To support this, the Brie architecture consists of a prototype-based language extension to Squeak with flexible inheritance, demand-driven evaluation, dependency-directed backtracking, and a special copy algorithm that conditionally copies dependent references.

1. Introduction

The Croquet platform [1] enables new dimensions of online collaboration by allowing users to create their own virtual environments, and to spontaneously meet with groups of users in these spaces. An obstacle to fully realizing this vision is the prohibitive cost of developing 3D content using the tools currently available. As described more fully elsewhere [2], Brie's solution to this problem is to empower users who would like to develop worlds, but are dissuaded by various limitations of current development tools.

The companion paper [2] provides the motivation and use case for Brie, and is a prerequisite to this paper. The other paper describes the four types of Brie/Croquet users:

- Consumers
- Authors
- Programmers
- Wizards

This paper describes the architecture and implementation of Brie, and will be of interest mostly to Programmers and Wizards. Although our strategy is to focus on Consumers and Authors, Brie's design makes some effort to insulate Programmers from the complexity of the core Croquet architecture.

Like "Smalltalk", "Brie" denotes both a programming language and the environment that language exists in. Brie implements a prototype-based language extension to Squeak, and a 3D direct-manipulation interface to objects created using this language. The 3D interface is designed to support content creation and reuse through interactive deconstruction and construction in a persistent, collaborative setting [2]. In order to support the direct manipulation of all kinds of objects, including behaviors, we must make them concrete. Thus the language is designed to support the reification of the primary aspects that a user might want to operate on. We say that such aspects are "Brieified," and the language is designed to shield Consumers, Authors, and even Programmers from bookkeeping tasks arising from interactions between Brieified components.

We describe the core characteristics of the Brie language, as well as an outline of its implementation. One key feature is the transparent integration of dependency-directed backtracking with Brie/Squeak syntax, so that state invalidation and recomputation occurs automatically without explicit direction from the programmer. Brie does not attempt to be a general-purpose wholly graphical language. For example, there

are no primitive behaviors for sequencing other behaviors, nor for conditionals. Only nullary "getter" and unary "setter" methods are provided for. If a Programmer wishes to create a new behavior from scratch, they must do so by writing text-based source code. Nonetheless, we will show that Brie achieves its goals.

2. The Brie Language

This section describes the major features of the Brie language, and how its implementation is integrated with the Squeak language. Although Brie is conceptually a pure prototype-based language, we create class hierarchies in order bootstrap the system and to utilize the programming tools available in Squeak. Hence, new types of objects are implemented as subclasses of BrieObject, new states as subclasses of BrieState, and so on.

2.1. Objects (BrieObject)

BrieObject is the fundamental Brie object type. It has two main responsibilities: implementing a customizable behavior-dispatch mechanism, and defining appropriate semantics for deep-copying. Copying will be described in a later section, once the requisite material has been covered.

Brie overrides the `#doesNotUnderstand:` method, which the virtual machine sends to the receiver of a message when Smalltalk method lookup fails to find a method matching the message selector. Rather than opening an error window, Brie uses the original message as input to its own behavior-dispatch algorithm. Each BrieObject maintains a dictionary of behaviors, keyed by selector name. If a BrieObject cannot find a matching behavior in its own dictionary, it can delegate the message to another BrieObject. If there is no object to delegate to, an error message is raised. For example, the programmer might arrange delegation to follow a chain of parents in Croquet's 3D scene graph. Thus Brie supports "Part-Whole" (aka "Has-A" or "Containment") inheritance, as well as the usual "Kind-Of" (aka "Is-A" or "Superclass") inheritance. In the default 3D interface application of the language, BrieObjects inherit from the assembly they are part of, then from the space that the object is in, and ultimately from the "interactor" or UI that the user is looking through [3]. The interactors described in [4] are an example of the latter. This allows BrieObject behavior dictionaries to be fairly small, with much of their behavior being provided by context.

We would like to emphasize that that this is but one possible implementation of the Brie semantics. It is conceivable to instead use Uniclasses, as done for EToys. However, such an approach would require more effort to implement. As we develop the Brie semantics, we value a design that is flexible and easy to implement.

2.2. Behaviors (Briehavior)

Brie behaviors, or Briehaviors, implement the methods `#invoke` and `#invoke:`. One of these is sent when the Briehavior is found by the dispatch algorithm described above; the one chosen depends on whether the original message has zero or one arguments. (Recall that only setter- and getter-like messages are supported. See "Introduction", above.)

Briehavior is a subclass of BrieObject. This implies a metacircularity, where each behavior has its own behavior dictionary, and therefore can utilize other behaviors attached to it. One consequence of this metacircularity is that initialization of the behavior dictionary must be demand-driven; otherwise it would recurse infinitely as each behavior has its behavior dictionary filled with behaviors that must then have their behavior dictionaries filled. To avoid this, the behavior dictionary of a newly instantiated BrieObject is initially empty, and is initialized on a per-behavior basis: when the behavior lookup mechanism fails to find a behavior in the dictionary, if first consults a prototype dictionary that contains the default behaviors for that type of BrieObject. If no behavior is found, the message is delegated to the next object in the chain. Since Briehaviors may have state, a Briehavior found in the prototype is copied before being inserted into the normal BrieObject dictionary, and then invoked.

The code examples in the next section show that once a behavior has been added to an object, invoking it looks just like a Squeak message send.

2.3. States (BrieState and BrieComputedState)

States are a type of behavior that plays a central role for Brie programmers. By integrating dependency-directed backtracking [5], we ease the bookkeeping required for creating highly interactive applications. In this section, we use a series of code examples to show the utility of this technique before outlining our implementation.

Note that this paper is about Brie internals. A typical interactive user would work only with graphical objects – including, perhaps, behaviors reified as buttons, menu items, or name/value pairs in an information-panel.

2.3.1. Getting and Setting State.

Our first look at Brie code shows how to add states to an object, and how to get and set the values of these states.

```
"Create a new BrieObject."
box := BrieObject new.
"Add new states to behavior dictionary."
lengthState := BrieState named: #length.
widthState := BrieState named: #width.
lengthState attachTo: box.
widthState attachTo: box.
"Typically, state is set this way..."
box length: 5.
"But we can also set it like this (as
is done by the dispatch machinery):"
widthState invoke: 7.
```

2.3.2. Computed States.

The purpose of computed states is to compute and cache some value based on the values of other states. This is done by creating a subclass of BrieComputedState, and overriding its `#computeState` method to compute the value to cache. Continuing with our example, we introduce a state to compute the side area of the rectangle.

```
BrieStateArea>>computeState
  "Compute area. 'receiver' is a pseudo-
variable that is bound to the object that
received the message."
  ^ receiver width * receiver length
```

Now, we add an instance of this state to our box.

```
areaState := BrieStateArea named: #area.
areaState attachTo: box.
"Computes, caches, and answers 35."
box area.
"Answers cached value of 35."
box area.
```

BrieComputedStates do not compute and cache a value until they are invoked; *`nil`* is stored in the cache to denote that *`#computeState`* must be invoked before returning a value. This "demand-driven evaluation" is important in dependency-directed backtracking.

2.3.3. Evaluation Context.

As in Self [6], behaviors may be found in other Brie objects. (See "2.1 Objects (BrieObject)," above). It is often necessary for the behavior invocation to know who the original receiver of the message is. The behavior dispatch mechanism provides this information by managing the binding of the `receiver` pseudo-variable (which we implement as an instance variable of Briehavior).

In addition, since behaviors are first-class objects, they may have their own state, methods, and behaviors. The computation for a behavior may need to reference these locally. Thus `self` designates the behavior object itself, not the receiver of the message. Both bindings are available to methods that implement a behavior. In the example above, within the body of `BrieStateArea>>computeState`, `receiver` is bound to `box`, while `self` refers to `areaState`.

2.3.4. Dependency-Directed Backtracking.

Given the ordinary-looking Smalltalk code above, how does setting the length of the rectangle invalidate the cached area? We have integrated dependency-directed backtracking into our behavior invocation mechanism [5]. The general idea is that invoking the length and width states during computation of the area causes them to record that the area depends on them. When they change, they invalidate all dependent states. This invalidation can propagate. For example, if some other state depended on the rectangle's area, then it would also have been invalidated when the rectangle's length changed.

```
BrieStateVolume>>computeState
   ^receiver area * receiver height.

(BrieStateVolume named: #volume)attachTo: box.
(BrieState named: #height) attachTo: box.
box height: 2.
box volume. "Computes and caches 70."
"Setting length invalidates area and volume."
box length: 6.
box area. "answers 42"
box volume. "answers 84"
```

Note that the system handles dependency fanout in both directions: area is dependent on and invalidated by a change to either length or width, and a change to length will cause both area and volume to be invalidated. This automatic updating makes the elements of an application act like a spreadsheet that is not limited to a 2D grid of cells.

In order to support dependency-directed backtracking, each BrieState has two bookkeeping fields, `usedBy` and `requires` that are used to represent a bidirectional dependency relationship between a pair of states. We maintain the following invariant: a state is `usedBy` another state if and only if the second state `requires` the first.

The `usedBy` field records all of the other states that used this state to compute their values. Above, `BrieStateVolume>>computeState` causes itself to be recorded in the `usedBy` field of both the area and height. Note that Programmers do not have to do this explicitly because Wizards have arranged for the behavior invocation process to do this bookkeeping. Whenever a BrieState is set, it invalidates all states that

it is `usedBy` so that they will be recomputed when next accessed. This process is recursive, since each invalidated state must similarly reset all states that it is `usedBy` (as in our example, where setting the length resets the area, which resets the volume).

The `requires` field points in the other direction, at all states that were used by this state to compute its value. When the state is explicitly set or reset, it removes itself from the `usedBy` field of each state that it `requires`; it will not later be invalidated by changes to these states. For example, consider a button that has a BrieComputedState that computes its color to be a shade lighter than the color of the window it is embedded in. When the color of the window changes, the button color will be invalidated and recomputed the next time it is needed. However, if we explicitly set the button's color to be red, the button will be removed from the window's `usedBy` field; changes to the window's color will no longer affect the button's color. If we then reset the button's color, the next use of it will again compute a value based on the window's color, which will reestablish the dependency of the button upon the window.

Invalidating a BrieComputedState is equivalent to setting its value to `nil`; the `usedBy` and `requires` fields are updated just as if the state had been set to any other value. As we described, BrieComputedState is demand-driven: the new value will not be computed (and cached) until some other object requests the value. Once the state has been reset, subsequent invalidations are cheap, since `usedBy` and `requires` are empty, and therefore no recursive invalidations are triggered.

There exist some "Wizards-only" subclasses of BrieState in which `#computeState` is eagerly evaluated immediately upon reset. Their implementations of `#computeState` include side-effects that are used for interfacing Brie to non-Brie objects. (For example, this is used in rendering.)

The current implementation does not detect cycles among the dependencies, although the information is available in the model and it would be useful for non-programmers if presented in the right way. We do intend for casual users to pull copies of computed behaviors out of one object and place them in another object. Circularities will happen.

Neither do we yet have language-level support for groups of mutually dependent values, of which one is expected to be supplied directly by user activity. For example, a sphere size could be specified by either one of radius or diameter, or a right triangle can be specified by any two of its side lengths.

3. Copying BrieObjects

Since Croquet allows objects to be copied into a world that is replicated on a completely different set of computers than it originally resided on, copying an object must recursively copy all references that the object needs to function properly. To accomplish this goal, Brie implements a dictionary-based 2-pass copy algorithm.

3.1. Motivation

Two passes are necessary because it is not always possible to determine when a variable should be left alone and when it should be rebound to a copy of its referent. As the following example shows, sometimes a variable should be rebound only if its referent has already been copied during the copying process.

Consider a device with a button that, when clicked, changes the device to a random color. If we copy the device, we would expect that clicking on the copied device's button would change the copied device's color. But if we only copy the button, then we would expect the copied button to act on the original device. In Brie, we implement this by only copying the button's device in the first case (when it needed to be copied anyway). We call this a "dependent copy variable", as opposed a "forced copy variable" which must always be copied.

3.2. Algorithm

Before starting to copy an object, a dictionary (the "copy map") is initialized; its role is to map each original object to its copy. In the first pass, an object told to copy itself first checks the copy map; if it has already been copied, the copy is immediately returned. Otherwise it copies itself (recursively performing any forced copies), adds the copy to the copy map, and returns the copy.

The second pass iterates over all copies in the copy map. For each object, it checks all dependent copy variables to see if a corresponding copy exists in the map; if so, the copy is assigned to the variable.

3.3. Interaction with Backtracking

When a BrieState is copied, we must maintain the `requires`/`usedBy` invariant (described in section 2.3.4) in a manner consistent with the desired semantics.

Consider two states U and V, such that U requires V, and therefore V is usedBy U.

3.3.1. Only Value Copied

If a value V is copied but its user U is not, then the copy V′ is not used by U. A change in V′ will have no effect on the original U that required the original V. The dependency relationship between U and V is unchanged, and the new copy V′ is independent.

3.3.2. Only User Copied

If U is copied but V is not, then the copy U′ has its own dependency relationship with the original V. A change in the original V will effect the copy U′. U′ requires V, and V is usedBy both U and U′.

3.3.3. Value and User Copied

If both the V and U are copied, then U′ requires V′ and V′ is usedBy U′. A change in V will effect U but not U′, and a change in V′ will effect U′ but not U. The copies are independent (dependency-wise) from the originals.

4. Mobile Code

In the larger vision of the Croquet Architecture, code is just another form of media [1]; this is known as mobile code. While the current version of Croquet does not yet support mobile code, Brie provides a workaround. Although new Squeak classes cannot be created and shared among members of a collaboration, instances of existing classes can be created and manipulated. Therefore, as long as we use existing types of Briehaviors and other BrieObjects, we can compose them as we see fit to gain the benefits of mobile code.

5. Even Later Binding

Smalltalk is a late-bound language because it is only during method dispatch that it is known whether a message is understood. However, a typical Smalltalk program does not create new code or change the structure of code in an application. In this sense, Brie is later-bound, since it is intended for code to be interactively changed through direct manipulation

6. Related Work

Brie's prototype-based approach is inspired by Self [6]. Dependency-directed backtracking was drawn from experience with high-end, rule-based CAD systems.

7. Conclusion

We have explained the technical details of the Brie platform, and described the motivation behind the design decisions that we have made. Brie is both a programming language as well as a framework for direct manipulation UIs.

8. Future work

We are currently developing "Open Implementation" protocols [7] for method dispatch and for multiple instance inheritance (including bidirectional inheritance).

Brie is intended to support the construction of new objects by assembling (e.g., dragging) objects and behaviors onto one another. It will be important to recognize when such operations are allowed, or what must happen to an object or behavior before it can fulfill its proper role in its new home. We plan to examine the use of behaviors on behaviors to define a protocol template that specifies what a Brie object must have in order to function in a given relationship. This template can be used to either add new functionality to the new component to ensure its success, or to indicate how the change is not appropriate.

In addition to the language-level constructs described here, we are using the Brie architecture to create a 3D, collaborative, direct-manipulation user interface [2]. It remains to be seen what other language-level features are needed to support this.

9. Acknowledgements

This work was undertaken under the direction of the Division of Information Technology of the University of Wisconsin-Madison. It was funded in part under a contract with the National Institute of Information and Communications Technology (http://www.nict.go.jp).

10. References

[1] Smith, David A., Alan Kay, Andreas Raab, and David P. Reed, "Croquet – A Collaboration System Architecture." *Proceedings of the First Conference on Creating, Connecting, and Collaborating through Computing (C5 '03)*, IEEE Computer Society Press, 2003.

[2] Stearns, Howard, Joshua Gargus, Martin Schuetze, and Julian Lombardi. "Simplified Distributed Authoring Via Component-based Object Construction and Deconstruction in Collaborative Croquet Spaces", Proceedings of the Fourth Conference on Creating, Connecting, and Collaborating through Computing (C5 '06), IEEE Computer Society Press, 2006.

[3] Smith, David A., Andreas Raab, Yoshiki Ohshima, David P. Reed, and Alan Kay. “Filters and Tasks in Croquet”, *Proceedings of the Third Conference on Creating, Connecting, and Collaborating through Computing (C5 ’05)*, IEEE Computer Society Press, 2005.

[4] Kadobayashi, Rieko, Julian Lombardi, Mark McCahill, Howard Stearns, Katsumi Tanaka, and Alan Kay. “3D Model Annotation from Multiple Viewpoints for Croquet”, *Proceedings of the Fourth Conference on Creating, Connecting, and Collaborating through Computing (C5 ’06)*, IEEE Computer Society Press, 2006.

[5] Stallman, Richard and Gerald J. Sussman. "Forward Reasoning and Dependency-Directed Backtracking in a System For Computer-Aided Circuit Analysis." Artificial Intelligence, 9:135-196, 1977.

[6] Smith, Randall B. and David Ungar. "Programming as an Experience, The Inspiration for Self." in J. Noble, A Taivalsaari & I. Moore, eds, "Prototype-Based Programming: Concepts, Languages, Applications." Springer-Verlag, 1997.

[7] Kiczales, Gregor, Jim des Reivieres and Daniel G. Bobrow. "The Art of the Metaobject Protocol." MIT Press, 1991.