Composition Ingredients

Design Review

and

Code Walkthrough

Author:

Lani Barton

Melora Svoboda

Reviewers:

Rob Jellinghaus

Create Date: March 10, 1997

Version: 1.0

Revision: 38

Document History

To automatically update the bottom row of this table, highlight the row and press the <F9> function key.

Version #	Rev #	Pages	Revision Date	Author of Revisions	Purpose of Revisions
0.0	1	5	03/10/97 9:46 PM	Lani Barton	Original document created from Lani's Notepad email attachment.
0.1	33	10	03/11/97 4:34 AM	Melora Svoboda	Added Figures. Edited and formatted document.
1.0	38	10	03/11/97 6:40 PM	Melora Svoboda	Incorporated Rob Jellinghaus' feedback and finalized document.
Last Saved	38	10	03/11/97 6:43 PM	Melora Svoboda	For each major set of changes, increment the Version # in the next available open row and type the updated info from this row into the other columns of the new row.

Table of Contents

Meeting notes 3/7/97

This was a code/design review of Rob's unum composition ingredient and presentation interface code, contained in these files:

ing_composition.plu

int_composition.plu

KEY DESIGN FEATURES

A key feature in Rob's design is using controllers and presenters in his code.

- Controllers are implemented by Harry's dynamics engine. They are Java objects which manage on-screen elements displayed by the renderer. They are implemented within our TCB (Trusted Code Base) and are accessed by unum code only via a limited Presenter interface. Thus, we refer to "presenters" which are actually controllers accessed through the protected presenter interface.

- Presenters are what Una use to display themselves. The ability to create presenters is ***the*** key point of TOS control with respect to on-screen display. Much of this design focuses on supporting secure creation and use of presenters via untrusted unum code.

ATTENDEES:

Pre-Break Moderator -- Brian

Post-Break Moderator -- Melora

Scribe/Editor -- Lani

Time Keeper -- MarkM

Educational Discussion/Walkthrough -- Rob

Reviewers -- Chip, Claire, MarkM, Crock, Trev

WHAT IS THIS CODE FOR:

This code deals with prop containment and presentation within a region. It dictates how a prop is allowed to be added to a region after an avatar has requested it, and the region does not know about this object. For example, an avatar might bring a gun in its pocket to a region, and want to take the gun out of the pocket once inside this region. To maintain consistent TOS, the region is responsible for validating presentations of an avatar and anything it contains.

HOW THE CODE WORKS

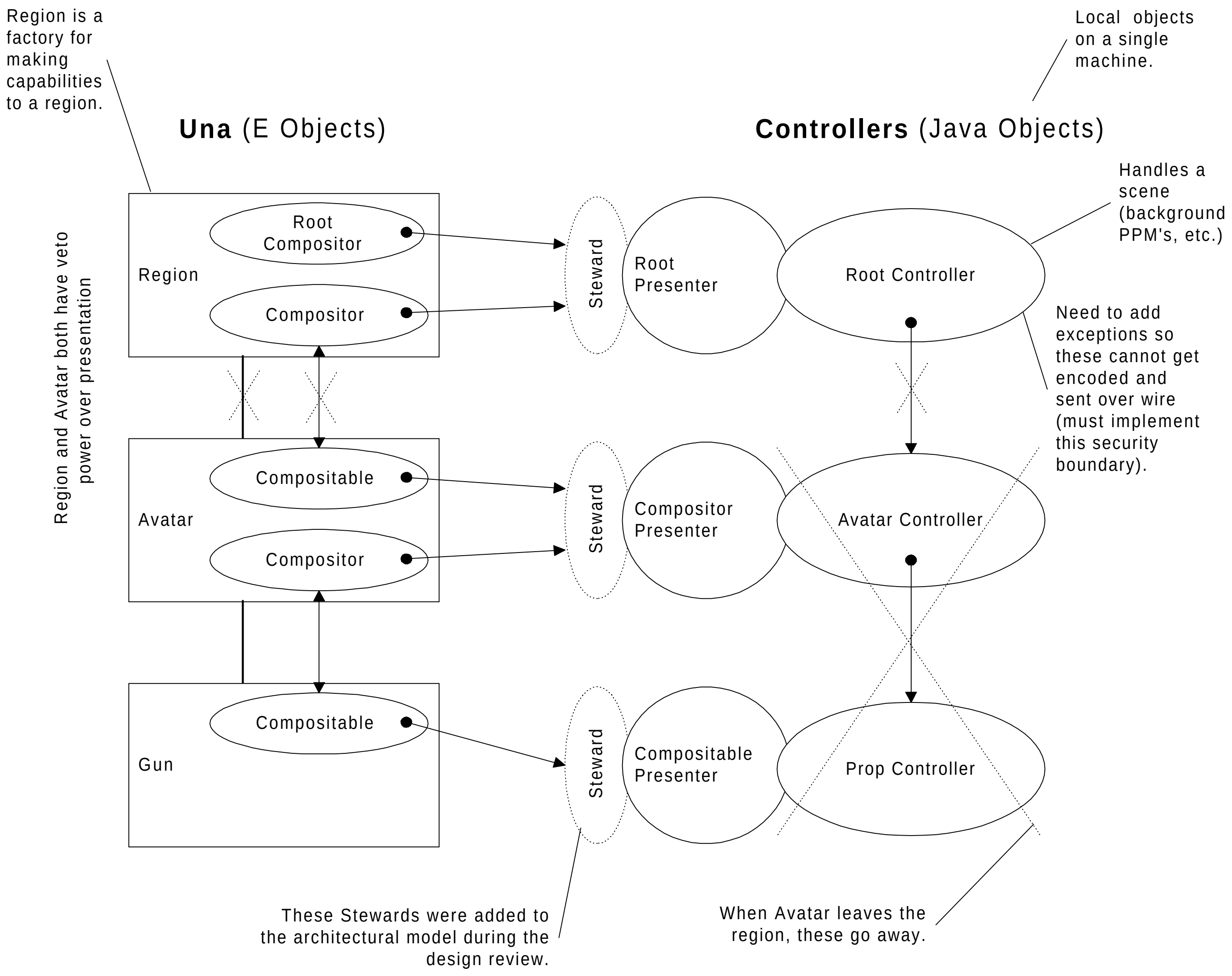
Region is a factory for making capabilities to a region.
Local objects on a single machine.
Una (E Objects)
Controllers (Java Objects)
Handles a scene (background PPM's, etc.)
Region and Avatar both have veto power over presentation
Region
Root Compositor
Compositor
Avatar
Compositable
Compositor
Gun
Compositable
Steward
Steward
Steward
Root Presenter
Compositor Presenter
Compositable Presenter
Root Controller
Avatar Controller
Prop Controller
Need to add exceptions so these cannot get encoded and sent over wire (must implement this security boundary).
These Stewards were added to the architectural model during the design review.
When Avatar leaves the region, these go away.

Figure 1: Compositor-Presenter-Controller Model

ASSUMPTIONS

Gun and avatar have already agreed

1. that the avatar wants this avatar, and
2. how this gun wants to be presented.

CAUSALITY FLOW

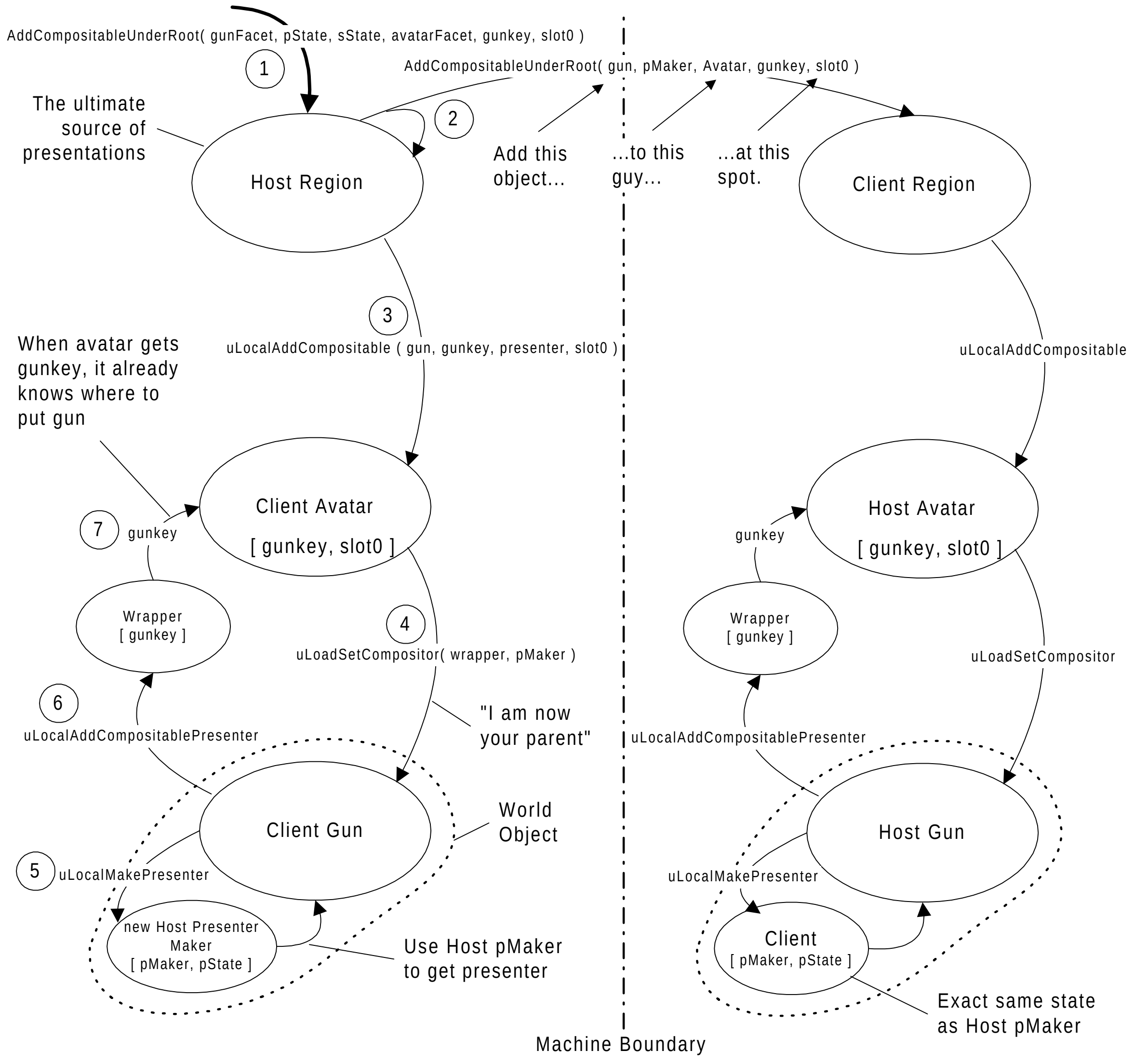
AddCompositableUnderRoot(gunFacet, pState, sState, avatarFacet, gunkey, slot0)
1
AddCompositableUnderRoot(gun, pMaker, Avatar, gunkey, slot0)
The ultimate source of presentations
2
Host Region
Add this object...
...to this guy...
...at this spot.
Client Region
3
When avatar gets gunkey, it already knows where to put gun
uLocalAddCompositable (gun, gunkey, presenter, slot0)
uLocalAddCompositable
Client Avatar [gunkey, slot0]
Host Avatar [gunkey, slot0]
7
gunkey
gunkey
Wrapper [gunkey]
Wrapper [gunkey]
4
uLoadSetCompositor(wrapper, pMaker)
uLoadSetCompositor
6
"I am now your parent"
uLocalAddCompositablePresenter
uLocalAddCompositablePresenter
Client Gun
World Object
Host Gun
5
uLocalMakePresenter
uLocalMakePresenter
new Host Presenter Maker [pMaker, pState]
Use Host pMaker to get presenter
Client [pMaker, pState]
Exact same state as Host pMaker
Machine Boundary

Figure 2: Causality Flow

Prior to the steps illustrated in **Figure 2: Causality Flow**, region finds out that a gun wants to be presented under an avatar. Region has decided that gun is OK and can be contained by the avatar within that region. Code then progresses through the following steps:

0. TOS ingredient of region sends `iAddCompositiableUnderRoot` to neighbor Root Compositor ingredient.

1. Region sends `pAddCompositableUnderRoot (` *gun, pMaker, avatar, gunkey, slot 0* `)` to itself and …

2. …its clients (fan-out occurs here), passing the ***gun***, ***presenterMaker***, specified ***avatar***, ***gunkey*** (from avatar), and ***slot*** location.

   - ❑ The ***presenterMaker*** is what makes a ***presenter*** for the ***gun***. This unum was created by the Root Compositor ingredient in step 0.
   - ❑ The ***gunkey*** is an ID that the avatar uses as a label and hash key for the gun. The gunkey distinguishes the gun from any other prop the avatar might be holding.
   - ❑ The ***slot*** is a location address (with respect to visual and spatial composition) for the gun.

3. Region sends `uLocalAddCompositable` (*gun, gunkey, pMaker, slot*) to its avatar presence, passing gun, presentation maker, gunkey and slot. Clients do the same to their avatar. This method lets the avatar know that it can go ahead and tell the gun it can be contained. Based on the value passed for the slot, avatar "remembers" the position where the gun is going to be added.

4. Avatar sends `uLocalSetCompositor` (*wrapper, pMaker*) to gun, passing the wrapper class and presentation maker of that avatar. Clients do same. This method tells the gun to "set me, the avatar, as your container – I am now your parent". The wrapper is an important security mechanism. Instead of interacting directly with the avatar, the gun deals with the avatar wrapper, which in turn deals with the avatar. This closes the security hole of the gun having direct access to its avatar; with direct access to the avatar, it would be easier for the gun to obtain its ID key , letting it present itself as anything it wanted.

5. Via the `uLocalMakePresenterAndGetState` method, gun uses presentation maker to make a presenter for itself. The presenter is the actual object that talks to the gun's controller. The presenter acts as a "steward", handling all presentation issues for the gun. This prevents the gun from faking its presentation state. The gun, together with its presenter-and-state vector, embodies a ***world object***.

6. Gun uses `uLocalAddCompositable` to send its presenter back to the avatar wrapper.

7. The wrapper then sends the gun's gunkey to the avatar. The avatar can now "hold" the gun within the region. The gun will be held in the slot which the avatar "remembers" from step 3.

Main feature is that the client presence of the pMaker gets state information from the <u>host</u> presence of the pMaker (<u>not</u> from the prop). The prop never asks the region directly for permission to be presented. This would force the region to retain too much state.

Also, the gun would have direct access to its key. That would be bad. Even if the gun made its own presenter, you could keep the key away from it, but you need to be thinking about this problem when implementing.

ISSUES

Rob's main issue here is to balance security of presentation and TOS controls, yet minimize the amount of state a region has to keep track of. The region should only deal with the things it immediately contains. This leads to the following issues.

- ❑ ***We want to avoid having the state of one machine moving to another just because an avatar does, especially if the TOS is the same*.**
- ❑ ***What happens when you remove an avatar from a region?*** In this situation, the entire structure (all the controllers) should also get deleted. This means that if avatar moves from region 1 to region 2, region 1 should delete all presentation of the avatar and its props. Deleting stale wrappers and presenters also resolves the issue of the prop not wanting to leave a region just because its container avatar does. (Don’t cache if you want a guarantee that state is not stale.)
- ❑ ***Chip asked why enumeration was used, rather than coding the presentation manager as a "factory object", like the UI controller?*** Rob thinks that's a good idea, and will resolve.
- ❑ ***What if a containership transfer happens during the presentation validation process?*** Rob needs to think through and resolve this.
- ❑ ***How do you avoid presence drift (that is, referencing an unum drags a lot of state info to a region)?***

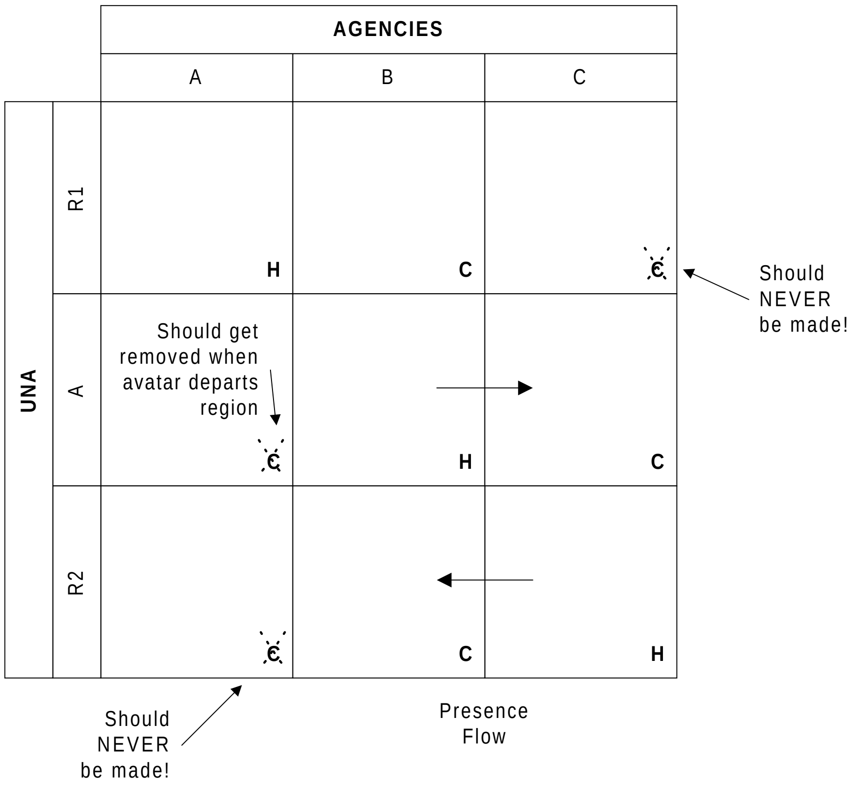

Figure 3: Avoiding Presence Drift

- ❑ ***Recursion.*** There are two scenarios: avatar in a region takes gun out of its pocket (this is covered by my current code) and avatar enters region while carrying gun in hand (this is NOT covered by my current code). Basically Rob's current code only supports a single object entering a region, whereas we also need to support an object-which-contains-other-objects entering a region. Rob knows how to do this but hasn't done it yet.

- ❑ ***Does this code meet its security claims?*** Rob will resolve.
- ❑ Mark had an issue dealing with ***quakes.*** Presenter maker, and all Una generally (e.g., gun, avatar, etc.) are "sturdy" objects, but others have no persistent state. There needs to be steward objects which implement the presenter interfaces. These presenter objects connect to the actual controllers. When there is a quake, the controllers fall away, and the steward presenter objects will need to recreate them during quake recovery. From the standpoint of unum code, the steward presenter objects are persistent and unaffected by a quake. Resolution?
- ❑ ***Synchronizing animation with containment transfer (that is, showing the avatar actually picking up the object).*** Rob was concerned about this, but Chip says time lag is negligible. No action needed.
- ❑ ***How much should presence cache its state vs. forwarding the state on to the new region?*** Pro for caching is that if there is sufficient "old" state data hanging around, a client region can calculate a local path for the avatar to pick up a prop based on this old information, and can send this to the host for validation. This saves load on the region host which doesn’t need to do path planning. Con for lots of caching is that cached info , especially mutable state, can be stale. However, since we don't assume global synchronicity with our model anyway, that can be OK.
- ❑ ***Another issue related to stale info - what if your avatar goes to pick up a prop, and another prop (from another client region) suddenly appears in the way and avatar walks into it?*** For now, avatar ignores new thing.

FALSE PATHS

- ❑ Closely held state requiring handoff of responsibility.
- ❑ Remembering recursive children which have been admitted.
- ❑ Unum programmers don't need to worry about handling quakes - the presenter steward developer handles it.

PATTERNS: WHERE THE CODE CAN GO WRONG

- ❑ For an object to have a trust relationship with another object, the first object needs to be told about departures (non-locally).
- ❑ Limit checking to "on the way in."
- ❑ Minimizing top level state and keeping in mind what it needs to be told about when changes occur.
- ❑ What happens in the event of local (but not system-wide) failure of pMaker? (i.e., the execution doesn't fail semantically, just resource-wise – NOTE: Controllers deal with race conditions.)
- ❑ What if while this is going on a containership change happens? ⇨ ***Need clean message ordering***.
- ❑ Revocability; singly and in aggregate.
- ❑ Don't cache if you want a guarantee that state is not stale.
- ❑ Use acceptable naming conventions.
- ❑ Think about control of presence spread. THIS IS A MAJOR SYSTEM-WIDE ISSUE!
- ❑ Avoid unnecessary mechanism. A suggestion for an ingredient design tips document (which we should write) is to add some unnecessary mechanism to the containership code (which currently is free of it), and then use it to show why unnecessary mechanism is bad.
- ❑ A suggestion for testing is to bring a simple prop (the lamp? flashlight?) into Microcosm, with simple host/client, to thoroughly exercise the composition ingredient code.