LUNAR LANDER GAME REVAMPED

INF221 SOFTWARE ARCHITECTURE ASSIGNMENT (Part 1 and 2)

SUBMITTED BY

SETH, PARUL
parul.seth@uci.edu

SREERAMAN, SREEVATSA
ssreeram@uci.edu

01/13/13

ABOUT THIS DOCUMENT

This document contains architectural models of a new version of Lunar Lander Game. This document has been made to capture the principal design decisions for developing the game. It consists of three sections. In the first section an overview of the game is given. In the next section, we have illustrated and explained three complementary models developed using xADL, Rapide and UML, and explained their consistency with each other. In the last section, we have analyzed our experiences and learning with respect to modeling, modeling notations and the tools.

GAME OVERVIEW

The application is a modern take on the popular lunar lander game. In this multiplayer networked version the game, the lunar lander has to deal with not just the physics of the universe but also protect itself from the attacks from the Lunar Base, which is continuously trying to destroy the lunar lander. Lunar lander also has a Mothership that is guarding it from the enemy attacks. This revamped Lunar Lander game consists of three players as described below;

1. Earth Mothership:

This is the central ship which protects the lunar lander from the base by sending fighters called attackers.

2. Lunar Lander

This is a ship which is landing on the surface of the moon. It also has defensive weapons which are affixed on its surface to ward off lunar fighters. These weapons are vulnerable to lunar fighter attacks and three hits will destroy the weapon/gun. The lander also has its health which will deteriorate due to hits from the lunar fighters or from the lunar base. It has limited fuel to land and if it has not landed before the fuel empties, it will fall and explode.

3. Lunar Base:

These are the forces that are based on Moon trying to stop the lander from landing on their home ground. The base can launch attackers to attack the lander and Mothership.

ARCHITECTURAL MODELS

We have developed three architectural models that show different viewpoints for the game design.

1. Structural Viewpoint in xADL

The model below represents the structural view of our game design. We have overlaid the C2 architecture with the client-server model. The client-server model used is the client being very thin and dumb and all the authority to make decisions rests with the game server. The reason for going with the client-server and not peer to peer architecture is due to security considerations of sharing IPs with one another. The reason for going with C2 is the separation of concerns achieved by C2 and the event-based nature of the system. The reason for making the clients dumb is that we think the synchronization would be very straightforward in this approach, compared to the server being relegated to the role of a blackboard where events are exchanged.

The C2 connector is standard i.e. it only allows requests to components above and notifications to components below. The network connector is the network which is connecting client and server.

The reason for replicating the models at the clients is to reduce the amount of information exchanged between client and server and only exchange the changes to the models and these changes will be applied to the models in the client which will be relayed to graphics library to be rendered.

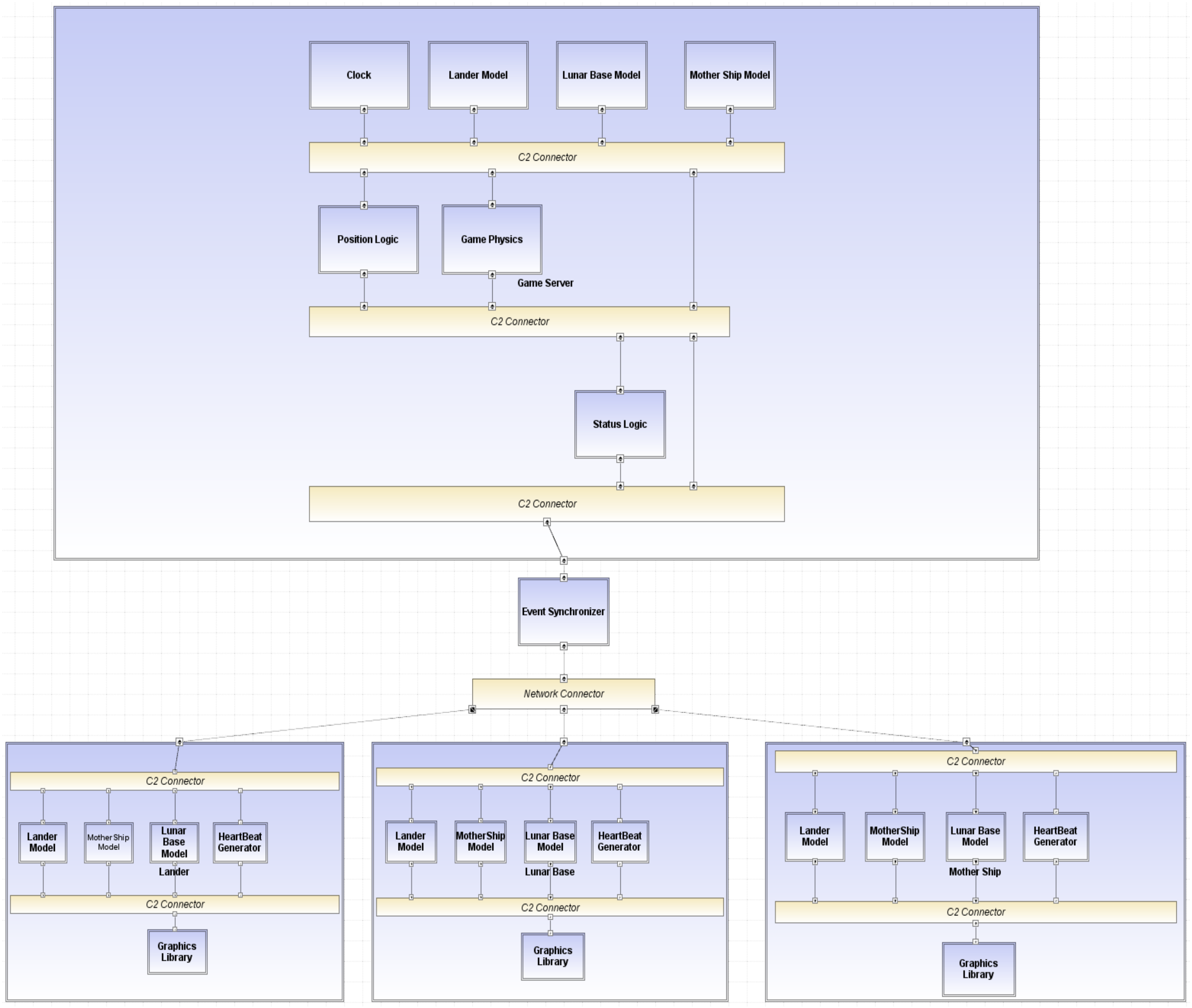

Figure 1 Lunar Lander Structural View in xADL

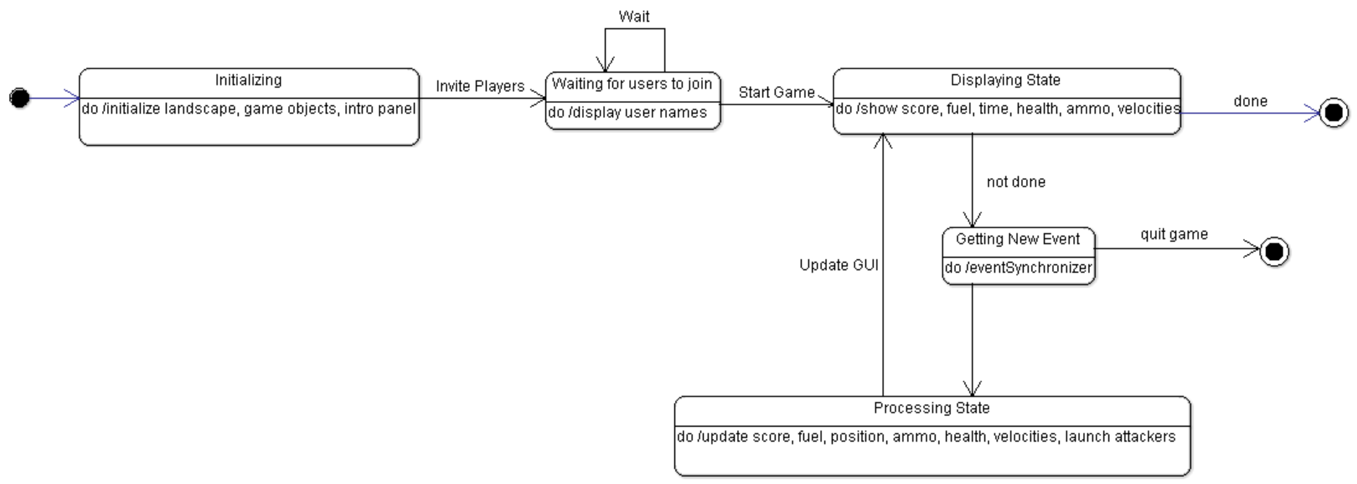
Wait
Initializing
do /initialize landscape, game objects, intro panel
Invite Players
Waiting for users to join
do /display user names
Start Game
Displaying State
do /show score, fuel, time, health, ammo, velocities
done
not done
Getting New Event
do /eventSynchronizer
quit game
Update GUI
Processing State
do /update score, fuel, position, ammo, health, velocities, launch attackers

Figure 2.1 Lunar Lander Game Statechart Diagram in UML

In the Figure 2.2 the statechart diagram of a lunar lander player is shown. In the first state a player joins the game as a lunar lander. Then in the next state player navigates the lunar lander by updating its position and burn rate. If the lunar lander is crashed or damaged or it runs out of fuel the game transitions to the next state where the player is destroyed and the game stops. Until then, a player can initiate attack in the next stage by choosing a target and launching ammo. The player continues transitioning between navigation and attack states until game ends.

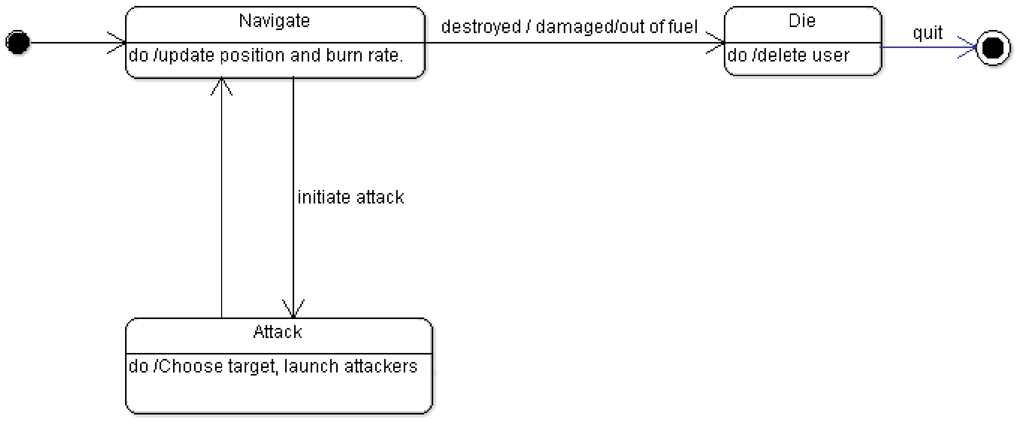
Navigate
do /update position and burn rate.
destroyed / damaged/out of fuel
Die
do /delete user
quit
initiate attack
Attack
do /Choose target, launch attackers

Figure 2.2 Lunar Lander Player Statechart Diagram in UML

While the above statechart diagrams are faithful representations of the system behaviors of both the game server and the lunar lander, it fails to offer adequate information pertaining to the interactions between various components. This is better represented with the help of a sequence diagram as shown in Figure 2.3.

This sequence diagram shows interaction between the game server, lunar lander, Mothership, lunar base, and attackers during a particular scenario of a game cycle. The lunar lander notifies its position and burn rate to the game server. Using this information the game server computes new position, horizontal and vertical velocities and fuel level of the lunar lander. It reports the updated position and fuel level back to the lunar lander.

Both the Mothership and the Lunar Base are capable of launching attackers to invoke an attacker. These components send request to the game server for invoking the attacker. Upon receiving the request the game server seeks the target position after receiving this information new attacker is created and it is instructed to move to the target location to the hit the target. The attacker follows this instruction and returns back to the base. In a concurrent scenario of lunar lander being damaged by an attacker, the lunar lander notifies the server that it has been damaged. The game server calculates the health of the lunar lander and reports it to the lander. If the health equals zero then the server notifies the lander, the lander calls its destructor and subsequently it is also deleted from the game. Similarly, other components can report damages, which are not shown in the sequence diagram for the sake of clarity.

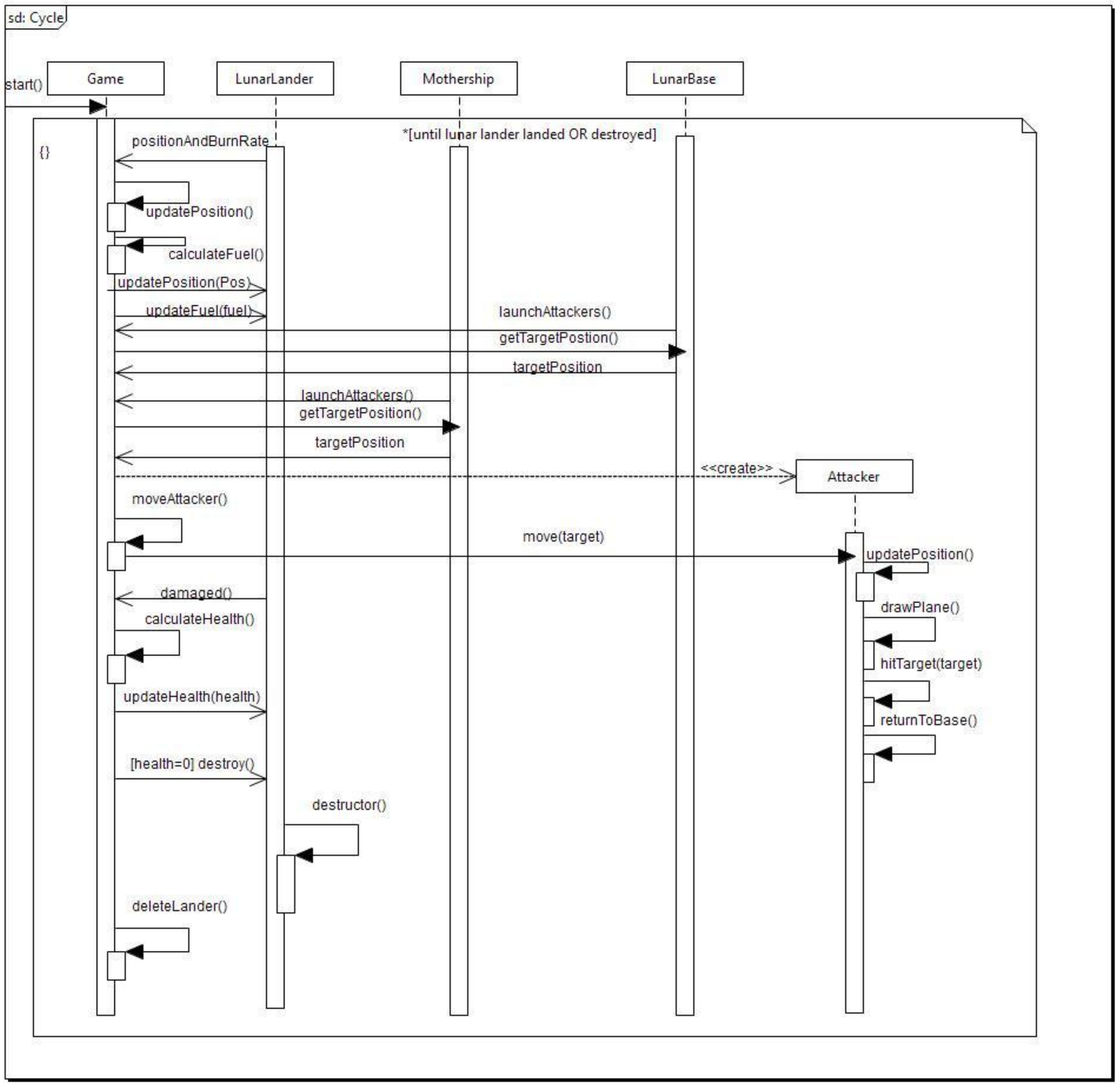

Figure 2.3 Lunar Lander Sequence Diagram in UML

3. Concurrency Viewpoints in Rapide

We use Rapide to show the concurrency viewpoint of our architecture. The primary objective of this model is to show the behavior of the event synchronizer and the interactions of the clients and server with the event synchronizer. The textual representation of the model is shown in Figure 3.

```
type Request is string;

type Client is string;



type GameServer is interface
action in  ProcessRequest();
           out  GenerateResult();
behavior
begin
   ProcessRequest => GenerateResult();;
end GameServer;



type EventSynchronizer is interface
action in    ProcessRequestWithHeartbeat(), RequestExecuted(),
Register(), DeRegister();
           out  ForwardRequest(), Notify();
behavior

          UnprocessedRequests: var Request[];

          action AddToUnprocessedRequests();

          action RemoveRequestFromUnprocessed();

          action AddToClientList();

          action RemoveFromClientList();

          action ClockTick();
          action PauseClients();
begin
        (ProcessRequestWithHeartbeat) => \

     if($UnprocessedRequests.length > 0 ) then \

          PauseClients(), AddToUnprocessedRequests(),
     ForwardRequest(); \

     else \

          AddToUnprocessedRequests(), ForwardRequest(); \

     end if;;

        RequestExecuted => RemoveRequestFromUnprocessed(), Notify();

        Register => AddToClientList();

        DeRegister => RemoveFromClientList();

        (?C in Client) ClockTick() => \

     if(NotHeardHeartBeat(?C)) then \

          PauseClients();

     end of;;

end EventSynchronizer;


type HeartBeatGenerator is interface
action  in    GraphicsEventGenerated();
           out  DispatchRequestWithHeartBeat();
behavior

     action ClockTick();

     action AddToRequest();

     action ConvertRequestToRequestWithHeartbeat();
begin

      GraphicsEventGenerated => AddToRequest();

      ClockTick => ConvertRequestToRequestWithHeartbeat(),
DispatchRequestWithHeartBeat()
end HeartBeatGenerator;
```

```
type GraphicsLibrary is interface
action  out  GraphicsEventGenerated();
end GraphicsLibrary ;


type GameClient is interface
action  out  DispatchRequestWithHeartBeat(), Register(), DeRegister();

     in    Notify();
end GameClient ;



architecture GameClientArch() for GameClient is
   GL : GraphicsLibrary;
   HBG : HeartBeatGenerator;
connect
      GL.GraphicsEventGenerated => HBG.GraphicsEventGenerated

      HBG.DispatchRequestWithHeartBeat =>
DispatchRequestWithHeartBeat;

end GameClientArch;



architecture BigPicture() is

  Lander, MotherShip, LunarBase : GameClient;

  EventSync: EventSychronizer;

  Server: GameServer;

connect

   Lander.DispatchRequestWithHeartBeat =>
EventSync.ProcessRequestWithHeartbeat();

   MotherShip.DispatchRequestWithHeartBeat =>
EventSync.ProcessRequestWithHeartbeat();

   LunarBase.DispatchRequestWithHeartBeat =>
EventSync.ProcessRequestWithHeartbeat();

   Lander.Register => EventSync.Register();
   MotherShip.Register => EventSync.Register();

   LunarBase.Register => EventSync.Register();

   Lander.DeRegister => EventSync.DeRegister();

   MotherShip.DeRegister => EventSync.DeRegister();

   LunarBase.DeRegister => EventSync.DeRegister();

   EventSync.ForwardRequest => Server.ProcessRequest();

   Server.GenerateResult => EventSync.RequestExecuted();

   EventSync.Notify => Lander.Notify();

   EventSync.Notify => MotherShip.Notify();

   EventSync.Notify => LunarBase.Notify();

end BigPicture;
```

Figure 3 Lunar Lander Concurrency View in Rapide

Consistency between Viewpoints

The section below compares different viewpoints illustrated in this document by showing the relationship and consistency between them.

Structural and Functional Viewpoint Consistency

The structural viewpoint shows all the elements that the system will consist including the components, connectors and the interfaces. The functional viewpoint depicts the actual interaction among these elements essentially the relationship between them. This is shown by models shown in figures 1 and 2.3. The underlying architecture i.e. client-server for the structural viewpoint as shown in Figure 1 is reflected in the sequence diagram of the functional viewpoint. The game server represents the server and the clients are represented by Lunar Lander, Mothership and Lunar Base. The clients send requests to the server as is shown by the interaction for the update position use case, the server then notifies the client of the changes made.

Functional and Concurrency Viewpoint Consistency

As depicted in the statechart diagram shown in Figure 2.1 showing the functional view, any event that is generated whether it is a new heartbeat or a user input goes to the event synchronizer. The actual concurrency management by the Event Synchronizer is modeled by the concurrency viewpoint.

Structural and Concurrency Viewpoint Consistency

The consistency here stems from the similarity in interfaces in the Concurrency view and the components in the structural view. While the structural view shows how the event synchronizer is statically placed in the overall design of things, the concurrency view explains what kind of behavior to expect from the event synchronizer and how the event synchronizer interacts with the clients and server. Again, while the structural view shows the placement of the heartbeat generator in the model, the concurrency view explains the interaction of the heartbeat generator with the synchronizer and the graphics library.

LEARNING EXPERIENCES

Modeling

Modeling as an activity encouraged us to think about the big picture view of the system as a whole. If we had started developing the game without a model, we would not have thought about synchronization and concurrency and incorporated it from the beginning. They would have been afterthoughts awkwardly pushed into code. Especially in cases where the developers are not familiar with the domain (as in our case) modeling helps them explore different aspects of the domain before starting to code.

Modeling Languages, Tools and Notations

We feel that the modeling language controls the extent of detail one goes into. This is not necessarily a negative aspect, but we feel that it interferes in the modeling approach. Compared to a whiteboard where the designer is in complete control over the modeling detail, where the designer can differentiate between what is a principal design decision and what is not, modeling languages seem to constrain this. The reason why I like Calico is that the designer can evolve the model, form their own notations and come up with designs without any constraints. The resulting designs may be idealistic or the programming languages and platforms may not support these decisions, but the tradeoffs made are rational and though the final system is not perfect, this design serves as a goal to strive for. The reliance on rigid syntax in modeling notations makes it unwieldy in the initial stages of exploratory design. The fluidity and freedom one experiences designing on a whiteboard is missing when using modeling notations.

Also, none of the modeling languages provide primary support for expressing **rationale** for design decisions made. The designer would need to resort to natural language explanation for this. For, example, in our architecture, we choose to replicate the different client models in the server and the client. But the reason, that we are trying to reduce the information exchanged between the game server and client to changes to the models cannot be expressed readily. We feel that this is one major drawback of ADLs since these architectures serve as documentation for software maintenance.

Initially, we decided on using UML, xADL and AADL as our modeling languages. But we finally ended up not using AADL. One reason was the longer learning curve and tool support (we tried working with OSATE 2 and Stood). Another reason was that after modelling the structural and behavioral aspects with xADL and UML we wanted to show the concurrency and synchronization management in our third model. Rapide served this purpose in a much better way, so even with the scanty documentation and no tool support available online we decided on using Rapide. That proved to be a good decision, considering the we had invested enough time in learning AADL without reaching to any proper conclusions, the time and learning curve for Rapide was shorter and much more useful.

Our experiences using the individual modeling languages and tools have been penned down below

UML

As a notation UML is easy to learn but it lacks flexibility and extensibility. Changes are hard to incorporate as compared to xADL. Also there is inconsistency among the notation across the literature available that creates confusion. UML has many tools available and that is both good and bad. Most of the tools that are available (we have mainly researched on free or open source tools) are not well grounded with UML specifications. The online tools for UML are more like diagram editors. We have previously (on job) used commercial tools for UML like IBMs Rational Rose and Microsoft Visio, they are definitely better than the free counterparts (for obvious reasons).

After searching a lot of tools we ended up using ArgoUML for the statechart diagram and Papyrus for the sequence diagram. ArgoUML was easy to use but since it lacked proper tool and documentation support for sequence diagram, we had to switch to Topcased. The Papyrus editor supported the specification in a better manner, but the tool can be improved in a lot of ways. Another flip side of using UML is that it does not have any good analysis tool. Modeling seems easy but verifiability is almost unattainable. We can make the UML diagrams but can never be sure of the appropriateness and consistencies of the models created for supporting the future implementation.

Rapide

One unique feature of Rapide is the ability to describe concurrency and synchronization behavior. The emphasis on interfaces, connections and architectures which describe the behavior of the interfaces is especially helpful in describing asynchronous interactions. The flipside of using Rapide is that, it feels like writing code and this may cause people to question using Rapide because if you are writing logic, you could as well write code. It is understandable that documentation is very sparse due to the absence of any activity in Rapide. But, on the learning side, one of the advantages of Rapide was that we had a very good example project based on the web, which showed how the model can evolve as multiple clients and complexity increases. Most of the syntax is based on that example (http://complexevents.com/stanford/rapide/examples/trw/index.html).

Also, there are no first class objects or structures i.e. no structures for data encapsulation. Only primitive data types could be found and this caused some difficulty in depicting a few interactions. Additionally, not having tool support hurts. But what hurts more is, even though we don't have graphical tools, what would have really helped is just an editor which can catch syntactic issues.

xADL (ArchStudio 5)

The tool support for xADL greatly helped in creating models (though there are glitches here and there). Compared to AADL, Rapide or even UML, creating the model was straightforward. The xADL reference gives the different schemas. Though this is helpful, the learning curve is quite steep. Having example "Hello World" projects which showcase the core competencies of the modelling language will be really helpful. This is where we found Rapide to be easier to learn, even though the documentation was sparse. The fact that ArchEdit and Archipelago had to be used as a combination (for example, for creating substructures) to create the model was kind of disconcerting. From our experience, a graphical editor would be used to model the higher level details and a ArchEdit-like editor would be needed to drill down into details of the model. But, creating substructures was a high level task and the inference that both editors have to be used in concert was non-trivial.