Using ATAM to Evaluate a Game-based Architecture

Ahmed BinSubaih[1], Steve Maddock[1]

[1] Department of Computer Science
University of Sheffield
Regent Court, 211 Portobello Street
Sheffield, S1 4DP, UK
Email: {a.binsubaih, s.maddock}@dcs.shef.ac.uk

Abstract. The paper examines the suitability of employing an off-the-shelf software evaluation methodology to evaluate a game-based architecture we have developed to aid game portability between game engines. We are interested in finding out if the goals of the architecture are met and which architectural decisions have contributed towards them and which ones have undermined them. The Architecture Tradeoff Analysis Method (ATAM) promotes itself as a tool with the capabilities of revealing these issues. In this paper we put ATAM to the test on our architecture and discuss the findings based on the outputs generated which include lists of risks, nonrisks, sensitivities, and tradeoffs made. The findings show that a game-based architecture can greatly benefit from using ATAM by revealing its strengths and weaknesses which can then be guarded and addressed respectively before evolving the architecture further. Additionally we present an Architecture Reactive View (ARV) to consolidate disparate outputs generated by ATAM into one which we consider as an improvement to ATAM.

Keywords: Architecture evaluation, game-based architecture, ATAM.

1 Introduction

Software architectures are comprised of three elements [5]: software components, externally visible properties of these components, and the relationships amongst them. In this work the software architecture we are going to evaluate has a component called "game space" that services a game to two different game engines: one is a commercial off-the-shelf game engine (Torque[1]) and the second is a bespoke simulation engine [8]. Using a game space allows adding a new game engine without the need to modify the game to suit the new engine.

Common approaches for conducting architectural evaluations are grouped into two categories [1]: questioning techniques and measuring techniques. The questioning techniques result in qualitative results and consist of scenarios, questionnaires, and checklists. In contrast the measuring techniques produce quantitative results by using metrics, simulations, prototypes, and experiments.

These approaches can either be carried out in an ad hoc way or a structured way. The ad hoc way does not have any obvious pattern other than randomly throwing challenges at the architecture and hoping that either the architecture can address them or otherwise they will reveal the architecture's limitations. Examples of architectures that used this way are: Testbed for Integrating and Evaluating Learning Techniques (TIELT) [4], MIMESIS [21], Gamebots [3], and the first prototype of the architecture being evaluated here [8]. The problems with the ad hoc way are:

- There is no guarantee that all the architecture's components are going to be exercised during the evaluation.

[1] http://www.garagegames.com

- Challenges thrown might be redundant since they probe the same architectural decisions.
- They tend not to articulate the key decisions made and the tradeoffs between them.
- They do not explicitly establish links between the quality attributes desired and the architectural decisions made.

The structured approach uses methods such as ATAM [13], SAAM [16], ARID [12], ABAS, PASA, and CBAM [5]. The method that stands out from these for having an output that corresponds to our needs is the ATAM, a method developed by the Software Engineering Institute (SEI). ATM comprises of 9 steps grouped into four phases: presentation (3 steps), investigation and analysis (3 steps), testing (2 steps), and reporting (1 step). It generates a number of outputs such as: a prioritised list of quality attributes, a list of architectural decisions made, a map linking architectural decisions to quality attributes, lists of risks and nonrisks, and lists of sensitivities and tradeoffs.
In this paper we chose the structured approach for three reasons. First, the aim was to establish a direct correlation between the architectural decisions we have made and their impact on the goals and quality attributes required. Second, the architecture we are evaluating is a prototype which is in its second evolution and we want to discover any risky decisions before evolving any further. Although ATAM is aimed at evaluating architectures after they have been designed but before they have been implemented, there are a number of projects on which it has been used on deployed systems such as the Battlefield Control System (BCS) and EOSDIS Core System (ECS) [13]. Third, we need to see which decisions are crucial to achieving our goals and need to be protected from future evolvements that might add, alter, or remove architectural decisions that might have adverse effects on our crucial decisions.
In section 2 we describe how a number of projects have benefited from using ATAM. Section 3 details a walkthrough of the ATAM phases carried out while evaluating our architecture. In section 4 we present the results and discuss the suitability of using this method on game-based architectures. Finally, section 5 presents our conclusions.

2 Related Work

The architectures that have used ATAM are command and control project [13], information systems (ECS [13], and Alexandria [2]). Although these architectures are general (i.e. not game-based architectures) they are described here since no examples were found of its use on game-based architectures. The command and control project applied ATAM to evaluate a Battlefield Control System (BCS). The system is used by army battalions to control the movement, strategy, and operations of troops in real-time in the battlefield. The quality attributes of the architecture are: modifiability, availability, and performance. After conducting ATAM evaluation it revealed some potentially serious problems to do with availability and performance. The availability was at risk because of the limitations of the backup approach used and the performance was very sensitive to the communication load.
The second project is the ECS whose role is to store data collected from satellites and process them into higher-form information to make them searchable. The amount of data collected is huge (approximately hundreds of gigabytes per day). A number of quality attributes were identified for the system such as: maintainability, operability, reliability, scalability, and performance. Among the findings it was revealed that availability was at risk because there was only a single copy of the databases and performance was sensitive to the overhead imposed by having to convert between various data types.
The last project is the Alexandria system which aims to provide a library of papers online. The quality attributes pursued for this system were: modifiability, availability, security, usability, performance, and portability. The evaluation showed that availability was at risk depending on

the database server chosen, as some require it to be offline when the data schema is changed. In addition the architecture performance was found to be sensitive to the security levels used.
Contrasting the findings from the two approaches (ad hoc and structured) it shows the superiority of the latter for the following reasons:

- ATAM provides clear articulation of the correlation between the architectural decisions and the quality attributes.
- ATAM identifies risks associated with each architectural decision.
- ATAM reveals sensitivities for which a slight change has significant impact on the architecture.
- ATAM identifies tradeoffs made which are decisions affecting more than one quality attribute [17].
- ATAM identifies nonrisks which are assumptions that must be held for these to remain as nonrisks. If changed these have to be rechecked.
- ATAM provides more direct probing of the architecture in the form of utility tree (shown later in Figure 4) which transfers ambiguous requirement statements to more concrete measurable scenarios.

3 Using ATAM: A Case Study

ATAM allows two different variants of emphasis when carried out. The first is architecture-centric and the second is stakeholder-centric. In the first the emphasis is on eliciting the architecture information and analysing the architecture. In the second the emphasis is on eliciting stakeholder points of view. These two approaches are called the two faces of ATAM. In this evaluation we adopt the architecture-centric face because the architectural decisions made are the main focus of the evaluation and because of the small number of stakeholders involved.

Phase 1: Presentation
There are three steps in this phase. The first step starts by presenting the ATAM process to the stakeholders. The second step defines the business goals of the architecture. For this architecture two business goals were specified. The first goal aims to break the current tightly coupled practices employed when developing games using game engines. The tight coupling occurs when developers put the game in the game engine's specific format making the game only available on that engine. The second goal is to make the architecture flexible to accommodate different games.
Finally in the third step the architect describes the architecture to the evaluation team highlighting the following: the quality attributes pursued, the architectural decisions made, and the overall architecture design. These are illustrated in the following sections.

Quality Attributes
The requirements document in a traditional system development cycle usually lacks or weakly articulates the quality attributes of the system according to [6]. Requirements document tend to be good at describing the functional requirements which are different from the quality attributes. The difference being that the system can have accurate functionality but does not deliver it on time which is a performance quality attribute. A quality attribute is characterized by three categories [13]: external stimuli, responses, and architectural decisions. The stimuli are the events that cause the architecture to respond. The way the architecture addresses these events must be expressed in concrete and measurable terms which form the responses. The last category is the architectural decisions that describe the approaches adopted and how they impact the quality attribute responses.
The architecture being evaluated here has three primary quality attributes, and they are in order of priority: portability, modifiability, and performance. The portability attribute aims to run the

same game on multiple game engines without modifying the game itself. The game referred to here is the plot as described by [9] "…game is really a story, and the game logic is its plot; all other elements we add are just media through which we tell the story." Figure 1 illustrates a conceptual overview of the architecture that can address this attribute by extracting the game from the engines and servicing it back to them. The portability attribute succeeds if the same game can be serviced to multiple game engines without any modification to the game.

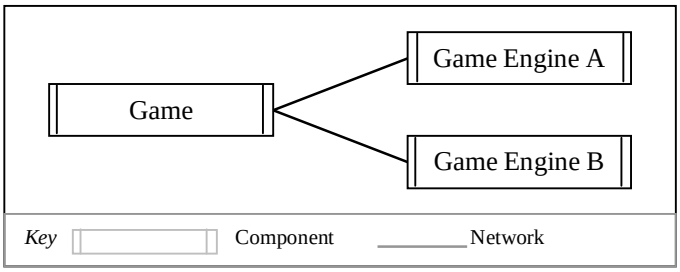

Figure 1: Game-based architecture that supports portability

The second attribute is modifiability which is the ability to make changes to the game easily. It is measured by the amount of changes required to the different components of the architecture and the less changes needed the better the modifiability is. The third attribute is performance which is the ability of the system to respond to stimuli within an acceptable timeframe. The stimuli can be user interactions or messages arriving over the network or method calls from other components or programs. The acceptable time to response for a stimulus is less than one second and the acceptable display rate at the game engine is no less than 20 frames per second (FPS).

Architectural Decisions

The success of an architecture is dependent on the achievement of its quality attributes and the forces at work behind these attributes are the architectural decisions. Getting these wrong could lead to a disastrous result. The remainder of this section describes the key architectural decisions made by the architect and organized by the three quality attributes they aim to support: portability, modifiability, and performance.

3.1. Architectural decisions to support portability

For the portability attribute the architect made three decisions with regards to how the game should be linked to the game engine. The three decisions are to remedy problems caused by practices adopted when developing a game using a game engine. To illustrate these problems we give an example of the two steps a game designer usually carries out to create a game:

- Step 1 creating the game environment (3D scene): a game designer uses the game engine's level editor to create the 3D environment using data created in modelling tools, sound tools, etc.
- Step 2 adding the game behaviour: which controls the game. This is added by direct access to the game engine's API or by using a scripting language or by using an editor such as the one provided by 3D GameStudio2 engine.

3.1.1 The three unwanted dependencies

There are three points that should flag warnings as far as portability is concerned when following the above steps for creating a game. These points relate to the dependencies made between the game and the game engine as shown in Figure 2. The dependencies marked with X are the warnings and the ones that need to be broken. The dotted line around the game space means that it does not exist physically on its own, but instead it lives inside the game engine. We put it on its own to conceptualise how it is linked to the game engine.

When a game is being developed using any game engine if that game engine asks for the game state to be put in its own game state that should raise the first warning flag. Instead what

[2] http://www.conitec.net/

developers should aim to do is to have the game state living outside the game engine's game state and find a way to communicate between the two states.

The second warning should be flagged when the game engine requires the objects to be represented in its own Game Model representation. The goal should be not to have a game model that is only accessible through the game engine's interface but have a game model that can be accessed outside the game engine. This could then be used by the behaviour engine to control the game by modifying the game state. The consequence of using the game engine's game model would mean that the manipulation of the game state would always be dependent on the game engine's game model interface which would correspondingly mean that it would have to be carried along with the game when moving to another game engine.

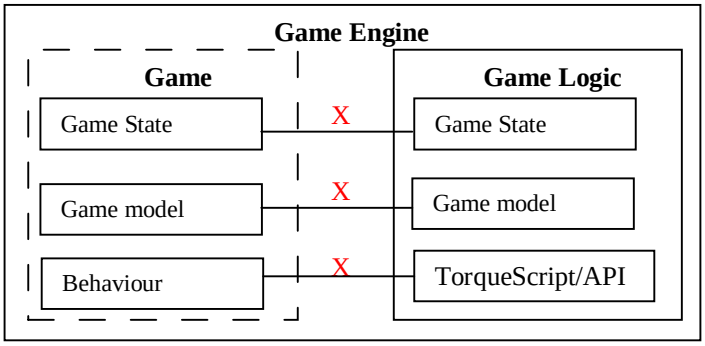

Figure 2: The three unwanted dependencies

The final flag should be raised when the game engine requires the behaviour to be specified in the game engine's own language or format. These three issues are the unwanted dependencies that make it cumbersome to port the game.

3.1.2 Architectural decisions to counter the three unwanted dependencies

Figure 3 shows the game-based architecture which consists of three subsystems. The game space, the adapters, and the game engines. The game space consists of game state, game model, and behaviour engine. The game space also has other components not shown in the figure like API, scripting interpreter, sockets, and persistent storage. These are used to manage the game and communicate it to the game engines. In the View part of the diagram it shows the game engine components which follow the decomposition scheme suggested by [18].

The decisions made to counter the unwanted dependencies are:

- *Dependency 1:* the direct link between the game space's game state and the game engine's game state should be broken. The architectural decisions made to achieve this were: model-view-controller (MVC) pattern (AD13)[10], asynchronous messaging (AD2), mid-game scripting (AD3) [19].
- *Dependency 2:* the direct link between the game space's game model and the game engine's game model must be broken. To achieve this the following architectural decisions were made: ontologies (AD4) [11], API (AD5), and mid-game scripting (AD3).
- *Dependency 3:* the game behaviour should not be formatted in the game engine's proprietary format. To achieve this the architectural decisions made were: API (AD5), mid-game scripting (AD3), ability to integrate specialized behaviour engines such as expert system engines (e.g. Jess [14]) (AD6), scripts mapping table (AD7), and objects mapping table (AD8).

3.1.2.1 AD1: Model-View-Controller (MVC)

The first architectural decision made was to adopt the MVC pattern. It was used to separate the system core from the view. In MVC the model contains the core functionality and system data. In this architecture that is placed in the game space. The view is used to display the information to the user, and in this case the game engine is the view. Lastly the controller handles the change requests by linking the view to the model. The adapters assume this role.

[3] Architectural Decision 1 – each architectural approach used is given a unique number for reference.

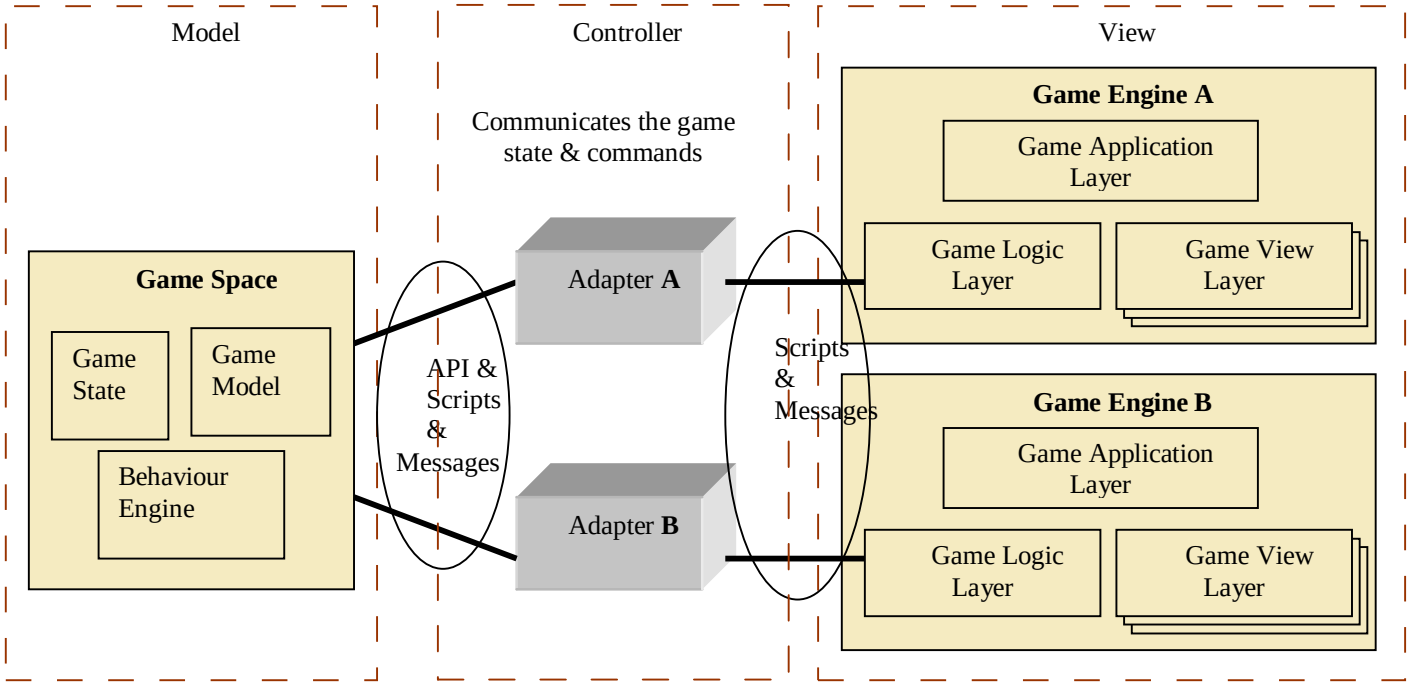

Figure 3: Game-based architecture using the MVC pattern

The main benefit of using this approach is that multiple views can exist for the same model and any modification carried out by one of the views is visible to other views. This is exactly what the portability attribute pursues (the game being the model and the game engines being the multiple views). However, the architecture does not follow the MVC pattern strictly as it breaks the direct link between the view and the model and only allows communication through the controller. This decision was deliberate to remove one of the known liabilities of using MVC which results in close coupling between the views and the model [10]. Therefore the approach can be considered as a variant of MVC. The corresponding elements to the game state's objects still have to be added to the environment and commonly this is done through the level editor of the game engine.

3.1.2.2 AD2: Asynchronous Messaging

The second architecture decision made was using messaging as a communication mechanism between the game space and the game engines via the adapters. More specifically the communication occurs between the adapter (which was decided to be on the game space side) and the game engine. The messaging role is to allow synchronization of the two game states (game engine's game state and game space's game state) and allow for actions triggered by the behaviour engine to be carried out to the game engine. The messaging mechanism used is asynchronous which aims at limiting the delay that is caused by the synchronous mechanism [20]. The messages arriving from the game engine are in a pre-determined format whereas the messages delivered to the game engine are formatted in its scripting language. If a specialized behaviour engine is used (AD6) messaging can be used to communicate between it and the game space. The format of the messages is dependent on the behaviour engine (see [8] for how messages were formatted for communication with Jess).

3.1.2.3 AD3: Scripting

The third architectural decision made was to use scripting to manipulate the game state on both sides. Scripting is a technique for specifying and manipulating the game without having to hard-code it in the system which makes it easier to modify. The scripting can be pre-compiled

(UnrealScript) or interpreted at run-time (known also as mid-game or on-the-fly scripting) using for example Python.

3.1.2.4 AD4: Ontologies

Ontologies are one of the architectural decisions made to counter dependency 2 with regards to the game model dependency break-up. Ontologies are used to specify the game knowledge representation independently from the game engine's game model. The main reason for using this approach is to avoid the inflexibility associated with building hard-coded classes. We wanted an approach that allows building classes on-the-fly. Ontologies allow creating concepts which have properties, and relations. We use these to create the classes of our domain. The concept becomes the class and its properties are the properties of the class and finally the relationships describe things like inheritance and association. The other benefits of using ontologies are: common language for describing a domain, machine interpreted language, and easy to understand and maintain.

3.1.2.5 AD5: API

The other decision to aid the game model dependency is to use an API to interact with the game space through code or through scripting. API and mid-game scripting are also used to set the game behaviour to break the last unwanted dependency.

3.2. Architectural decisions to support modifiability

The decisions taken with regards to modifiability aims to limit how far changes made to the game in the game space propagate to other parts of the architecture. The game space as shown earlier consists of three subsystems: game state, game model, and behaviour engine. Changes to any of these should be made with as little disruption as possible to other components in the game space including the adapters. The following sections describe the decisions made to address these concerns.

3.2.1 AD1: MVC

The first decision made to aid this is the use of the MVC pattern which aims to achieve this by separating the core of the system where the modification is going to happen from the view. Based on MVC any modification that happens to the game state is restricted to the changes in the game state model/class diagram and would not have ripple effects on the adapters (controllers) or the game engines (views). Nor will it affect the game model. However as far as its effects on the behaviour engine, it is dependent on how the behaviour is specified. If the added logic is already addressed in the behaviour by its game model type then there is no need to modify the behaviour engine. For instance, if the logic to be added is a non-player character (NPC) called "Kork" and the behaviour already says something like "If a NPC sees an enemy it should fire at it" then there is no need to modify the behaviour engine since the "Kork" is addressed by its game model type ("NPC") and hence the modification effect is confined to the game state. However, if the behaviour specifies something specifically for "Kork" and not for its game model type ("NPC") then it would have to be added and the effect propagates as far as the behaviour engine.

The use of MVC has a similar effect on any changes required to the second component (the game behaviour) as it can be carried out independently of the other parts of the architecture.

3.2.2 AD3: Scripting

Another key decision that enables better modifiability of the game state and game behaviour is the use of mid-game scripting. This enables manipulating the game state and setting the behaviour without recompiling the game space system. This also helps in making any modification required to the game engine without affecting the game engine code since you can access it at run-time. Mid-game scripting is used instead of pre-compiled scripting since using the latter would require some of the game logic to be put in the game engine's specific format as

was the case with Memesis [21] which used pre-compiled UnrealScript classes. The applicability of using scripting is dependent on the level of access given to the scripting. The level required varies from game to game and the minimum requirement is for a level of access which caters for the interaction required by that game.

3.2.2 AD4: Ontologies

Another modification that might be required is modifying the game model. This is the most expensive modification in terms of how far changes propagate. Changing the game model requires changing the behaviour engine, and the adapter. For example, consider a class called NPC which has a property called SeeHuman. If that property is changed to HumanVisible, then the behaviour engine that uses this property has to change so does the adapter if it uses it in communication with the game engine.

3.2.3 AD7: Scripts mapping table

Using a scripts mapping table is another architectural decision taken. The table acts as the translator that holds the same sentence in two languages: the one understood by the game space (Jython) and the one understood by the game engine (TorqueScript or Python). Each sentence holds placeholders for the information to be replaced at run-time.

3.2.4 AD8: Objects mapping table

The final architectural decision made is to use an objects mapping table. This is necessary to link the corresponding objects on both sides if similar unique identifiers cannot be set on both sides for the same objects.

3.3. Architectural decisions to support performance

While promoting portability and modifiability the architecture needs to perform at an acceptable speed. Since quality attributes do not exist in isolation from one another a compromise has to be achieved between the different attributes to reach a common acceptable goal. For instance to cater for better modifiability mid-game scripting was used despite the fact that mid-game scripting runs at much slower speed than the precompiled code (as much as 10 times slower).

The introduction of layers by using MVC to promote portability and modifiability also adds processing overhead on the architecture as information needs to be passed and sometimes translated between the layers. Moreover, the translation done in the controller which employs the Adapter pattern [15] to accomplish that by using a scripts mapping table requires processing overhead. The other overhead that affects performance is the network overhead because of the distributed environment used since the data has to be transferred across the network. The major reason for using a distributed environment is because game engines are known for making use of every last drop of processing power and thus running the game space on the same machine that is running the game is too much to ask. Additionally, doing so limits the possibility of having the architecture run different game engines simultaneously to communicate across the same architecture —this is an interoperability quality attribute for the future but this shows the benefits of at least listing the quality attributes as it forces the architecture to weight the impact of any decision on the present and futuristic attributes.

To compensate for the network overhead the architecture uses an asynchronous communication mechanism to reduce the impact on the display rate. The benefits of using asynchronous communication is that the calling system can continue processing after making the request and does not have to wait for a response [20]. This plays a vital role in keeping our FPS at the required level (20 FPS).

Another decision that affects the performance and relates to the network overhead is the placement of the adapters. The architect had to choose from three locations: placing the adapter in its own application on its own machine, placing the adapter in its own application on either the game space machine or the game engine machine, or placing the adapter in the same application as the game space. The first option was dismissed because it would have added another network

overhead. The second option was disregarded because it would have added extra overhead to communicate across applications. The third option had the least overhead expense and was the one chosen.

Phase 2: Investigation and Analysis

This phase comprises of three steps: identifying the architectural decisions made (step 4), generating the quality attribute utility tree (step 5), and analysing the architectural decisions (step 6).

Although in perfect documentation all the architectural decisions should be listed that is not always the case as was found when evaluating the BCS and ECS projects [13]. Therefore, the evaluation team has the responsibility to elicit architectural decisions from the architecture documentation, and the architect's presentation. This should elicit any architectural decisions not highlighted by the architect. Eight architectural decisions have been described above.

The utility tree elicits the quality attributes down to the scenario level to provide a mechanism for translating business goals into concrete practical scenarios. The utility tree also aids in prioritising the quality attributes. According to [13] this step is considered a crucial step which guides the rest of the analysis without which the evaluators could spend valuable time analysing the architecture without addressing the important quality attributes as far as the stakeholders are concerned.

Figure 4 shows the utility tree for the architecture. There are three levels in the tree: the quality attributes level, the refinements level, and the scenarios level. The aim of the refinement is to decompose the quality attribute further if possible. The last level holds the scenarios in a concrete form. This level also holds the scenarios' rankings which decide their priorities. In ATAM, ranking format can differ based on the participants' preferences. It could be a scale based on 0 to 10 or relative ranking such as High (H), Medium (M), and Low (L). The ranking is done using two variables: importance and difficulty. Importance states how important the scenario is for the success of the architecture and difficulty describes the degree of difficulty in achieving that scenario. We used the relative ranking to prioritise the scenarios.

ATAM heavily relies on scenarios to turn ambiguous quality statements such as "the system should be modifiable" into something more concrete as shown in Figure 4 by the scenarios: M1.1, M2.1,2.2,2.3, and M3.1. The benefits of using scenarios are threefold [13]. First, they are simple to create and understand. Second, they are inexpensive. Third they are effective.

The mechanism followed by ATAM to describe these scenarios uses a three-part format: stimulus, environment, and response. The stimulus describes what initiated an interaction with the architecture. The environment describes the state of the architecture when the interaction takes place. The response explains how the architecture reacts to the interaction. ATAM also encourages eliciting scenarios by thinking about different scenario types. The three types used are: use case scenarios, growth scenarios, and exploratory scenarios. The first describes typical uses of the architecture, the second lists anticipated changes, and the third lists extreme changes aimed to stress test the architecture.

Nine scenarios have been elicited for the architecture. The scenario elicited to prove that the architecture supports portability is described in P1.1 in Figure 5. This scenario requires the same game to be run on different game engines without any modification to the game. As described earlier the game here refers to the three main components of the game space. This scenario is ranked with high for importance because it is the main quality attribute for the success of the architecture. The difficulty rate is set to medium, as the change requires creating a new adapter between the new engine and the game space. The other scenarios elicited are shown in the utility tree.

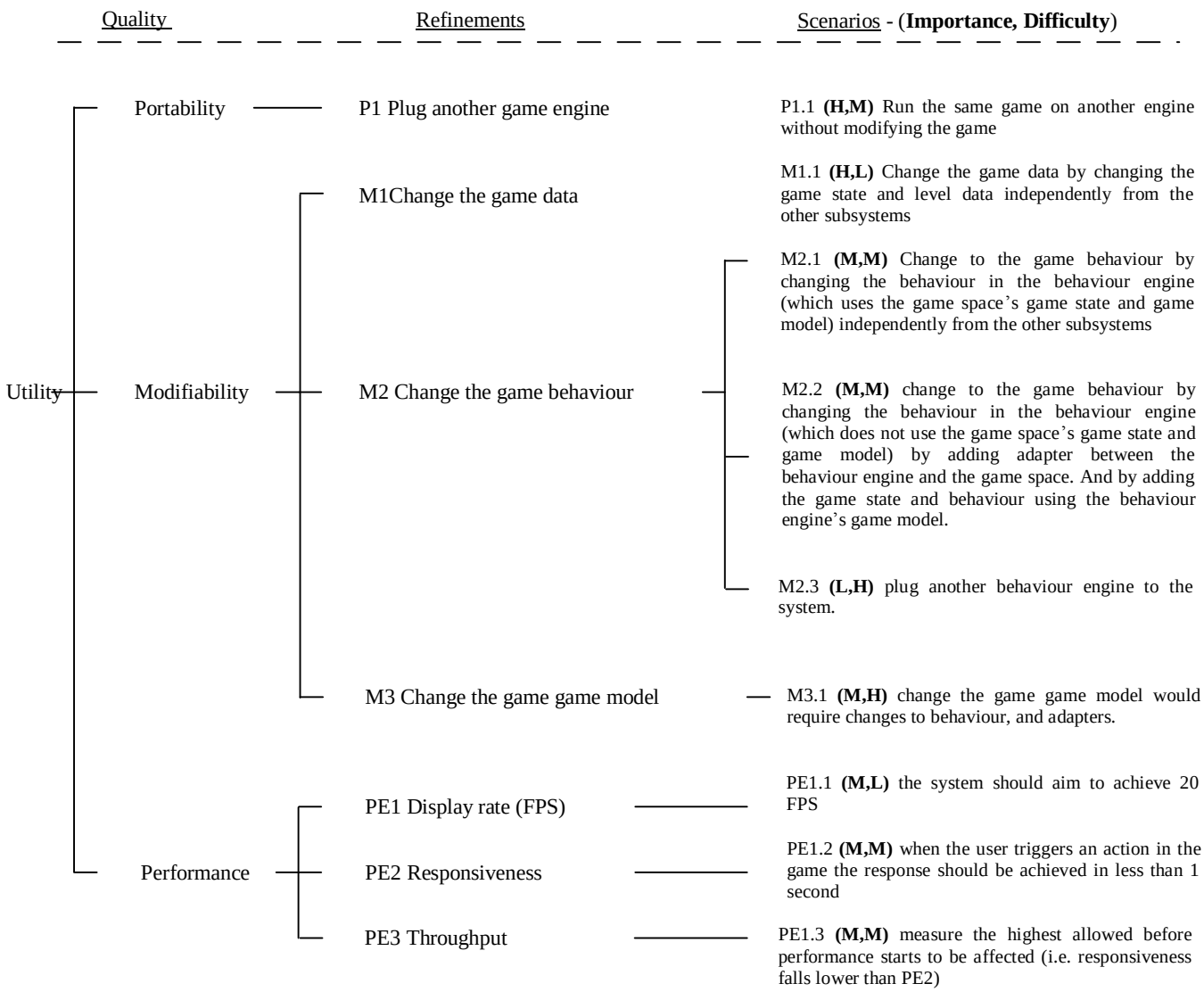

Figure 4: Utility tree

The last step in this phase is analysing the architectural decisions made to examine how well they correlate to the quality attributes. This step is where the architect's decisions come under close scrutiny. The output of this step is a detailed description of the decisions that are aiding the pursued quality attributes and the ones that undermine them. Table 1 shows the outcome of analysing the portability scenario for which a number of key architectural decisions have been identified. As illustrated in the table, for each architectural decision this step reveals sensitivities, tradeoffs, risks, and nonrisks which are described in Figure 5.

As shown in the table the first decision affecting this scenario is the MVC decision (AD1). MVC has one tradeoff, two risks, and one nonrisk associated with it. The tradeoff made (T1) favours portability over performance as described earlier. The first risk (R1) is caused by the tight coupling between the controller and the model which is a known liability of using this pattern [10]. The second risk (R2) is caused by the difficulty to maintain the data integrity between the two states. The nonrisk (N1) exists because of the decision taken to remove the other liability of using MVC – the removal of the direct link between the view and the controller described earlier.

The analysis of the messaging approach (AD2) has revealed two sensitivities, one tradeoff, one risk, and one nonrisk. The two sensitivities (S1, S2) are introduced because of concerns over network latency and messages load. Increasing either of these would have negative impact on the architecture's performance. The risk (R3) is introduced by the tight coupling as a consequence of using pre-determined messages for the messages arriving from the game engine which means

Analysing Scenario	P1.1			
Scenario	plug another game engine to the architecture			
Attributes	Portability			
Stimulus	running the same game on another game engine			
Environment				
Response	should be achieved by adding an adapter to connect the game space and the newly added game engine			
Architecture Decision	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
AD1 MVC		T1	R1,R2	N1
AD2 Messaging	S1,S2	T2	R3	N2
AD3 Mid-game scripting		T3	R4	N3
AD4 Ontologies		T4		
AD6 COTS behaviour engine	S1,S2		R2	N4

Table 1: Portability scenario analysis (see Figure 5 for description of S1, S2, etc)

Sensitivities:

- S1: Concern over network latency
- S2: Concern over messages load
- S3: In the event that a single unique identifier cannot be set this architecture decision becomes very sensitive to any modification as they have to be added manually to the table
- S4: Using ontologies allows for better game model scalability but it makes the architecture very sensitive to change as the change propagates to behaviour and adapters.

Tradeoffs:

- T1: Portability (+) and Modifiability (+) vs. Performance (-) – separating the architecture into layers adds an overhead for exchanging information between layers which affects the performance.
- T2: Portability (+) vs. Performance (-)
- T3: Portability (+) and Modifiability (+) vs. Performance (-) – mid-game scripting allows better portability and modifiability but runs at 10x slower than pre-compiled code.
- T4: Portability (+) and Modifiability (+) vs. Performance (-) – using ontologies allows for having a game model that is independent from the game engine's game model however it has adverse effect on performance.
- T5: Modifiability (-) vs. Performance (+)

Risks:

- R1: The risk is caused by the tight coupling between the controller and the model which is a known liability of using this pattern.
- R2: Data integrity
- R3: The risk is introduced by the tight coupling as a consequence of using pre-determined messages for the messages arriving form the game engine which means changes to them would require deployment of new system every time.
- R4: The risk raised is a consequence the game engine or the game space which might not have fully exposed their functionality through the scripting.
- R5: If no unique id can be set this means the mapping table should be done manually.

Nonrisks:

- N1: The nonrisk exists because of the decision taken to remove the other liability of using MVC – the removal of the direct link between the view and the controller described earlier.
- N2: The nonrisk risk exists because of using asynchronous mechanism as it avoids negative impact on the frame rate that could be caused by the synchronous mechanism
- N3: The nonrisk is the use of API approach which should stay compatible.
- N4: assume that the behaviour engine requires the game state to be replicated to its own working memory
- N5: A one-to-one mapping for objects across both game states is established by having a unique identifier and if that is not possible the objects mapping table handles that.
- N6: Should stay compatible

Figure 5: Sample from the analysis of sensitivities, tradeoffs, risks and nonrisks for Utility Tree in Figure 4.

changes to them would require deployment of new system every time. Using messaging middleware (such as publish-subscribe [10]) could improve the portability and loosen the coupling however it would have an adverse affect on performance and the tradeoff made here favours the performance. The nonrisk (N2) exists because of using asynchronous messaging as it avoids negative impact on the frame rate that could be caused by the synchronous mechanism.

Type	Scenario
Use case	- Run two games from the same domain - Run two games from two different domains
Growth	- Interoperate between two game engines - Run two behaviour engines (internal and external)
Exploratory	- Increase the load by increasing the number of NPCs the user can interact with - Increase the load by simulating multiple number of simultaneous players - Run the game space on the same machine as the game engine - Run the behaviour engine and the game space on the same machine as the game engine

Table 2: Scenarios generated in step 7

Mid-game scripting (AD3) has one tradeoff (T3), one risk (R4), and one nonrisk (N3). The tradeoff favours portability over performance since using scripting is slower than using pre-compiled code. The risk raised is a consequence of the game engine not allowing full exposure to its functionality through the scripting. The nonrisk is the use of API which should stay compatible as the API evolves.

The use of ontologies (AD4) has one tradeoff (T4) associated with it. The tradeoff is between portability and performance as ontologies improve portability but affect performance because the ontologies are not hard-coded as classes are.

The final decision that could affect portability is the use of COTS behaviour engine (AD6) which has two sensitivities (S1, S2), one risk (R2), and one nonrisk (N4). The sensitivities are similar to those caused by the messaging decision. The risk is similar to the second risk caused by MVC which relates to maintaining data integrity. The nonrisk assumes that the behaviour requires the game state to be replicated to its working memory.

Similar tables are created for each scenario listed in the utility tree. Figure 5 shows the sensitivities, tradeoffs, risks, and nonrisks from all the analysis.

Phase 3: Testing

This phase consists of two steps: brainstorming and prioritising scenarios (step 7), and analysing the architectural decisions (step 8). The first looks similar to generating scenarios for the utility tree in phase 2, however the aim here is different. There the stakeholders were asked to generate scenarios based on given quality attributes, here they are asked to ignore that and give general scenarios. The goal of this step is to widen the spectrum from which scenarios can be elicited. This step has been described as a bottom-up approach and the utility tree step as a top-down approach. Once the scenarios are generated the ones that address the same concern are merged together. Then the scenarios are prioritised. The prioritisation process here differs from the prioritisation approach adopted earlier. Here each stakeholder is given a number of votes (usually ATAM suggests 30 percent of the number of scenarios rounded up) and they use these votes to prioritise the scenarios.

After prioritisation, the scenarios generated are compared to the ones generated in the utility tree creation step. The goal is to plug these scenarios into the utility tree. While doing so one of three things could happen. First, the scenario might match an already-existing scenario and no further action is required. Second, the scenario will fall under a new leaf node of an existing branch; if so it is then placed under the branch. Third, the scenario might not fit in any branch and in that

case it means it is expressing a new quality attribute which has not been accounted for and thus the quality attribute is added to the tree. Table 2 specifies the scenarios generated in this step.
The next step in this phase is to analyse the architectural decisions. Similar tasks to the ones carried out in phase 2 are conducted on the new scenarios.

Phase 4: Reporting
The only step in this phase is presenting the findings to the stakeholders (step 9). The findings are usually summarized in a document containing the following outputs: architectural decisions used, scenarios and their prioritisation, utility tree, risks and nonrisks, and finally sensitivity and tradeoff points. One of the strengths of ATAM is that at the end of the evaluation process the results already exist since in each step the documentation is quite comprehensive. Thus, generating the final document becomes a simple task of merging the outputs of previous steps. Also suggested in this step is to generate risk themes by grouping risks discovered by some common factors. The risk themes generated for this architecture are:

- Performance is affected by the separation decisions made (MVC, scripting, and messaging) to achieve portability.
- The architecture relies on the ability to set a unique identifier for corresponding objects otherwise it severely affects modifiability.
- The data integrity across the different game states is at risk.
- There is a danger if the message load increases that the game space becomes the bottleneck in the architecture.

4 Discussion

The ATAM evaluation has delivered on its two main promised outputs: sensitivities and tradeoffs, and architectural risks as shown in Figure 5. The most important risks found were described in the risk themes (Phase 4). ATAM has also produced a list of the key architectural decisions used and classified them according to how they affect the architecture (support or undermine) by discussing their sensitivities, tradeoffs, risks, and nonrisks as shown in Figure 5.
Additionally ATAM completed a full circle by linking the architectural decisions to the quality attributes and back to the business goals. This is very beneficial to the stakeholders who are not technical to identify which architectural decisions have contributed or compromised their goals.
We found the evaluation process not only helpful to understand the architecture better but now it should also act as a guide when we need to modify or evolve the architecture. The guidance it gives is based on the revealed architectural decisions and their strengths and weaknesses. We also found the ability to choose one of the two faces (architecture-centric and stakeholder-centric) of using ATAM helpful because of the small number of stakeholders involved in this architecture. Another invaluable tool is the utility tree as it focuses and directs how the architecture should be examined by producing concrete measurable scenarios for otherwise ambiguous attributes.
The one issue we faced with ATAM is that it produces an analysis table (e.g. table 1) per scenario and as the number of scenarios grows it becomes difficult to conceptualise the whole architecture as a single artefact based on all the tables generated. For instance if we wanted to look up the quality attributes affected by an architectural decision we would have to examine all the tables to find that out. One possible workaround was to consolidate all the disparate tables into one single entity that can reveal this faster. We created a view for the architecture called the Architecture Reactive View (ARV) shown in Figure 6. From this view the architect gets the whole picture about the architecture as it describes the interaction between three elements: quality attributes, architectural decisions, and ATAM output. First, the architect can find out which architectural decisions affect which quality attributes. Second, he can also find which

ATAM output affect which architectural decisions. Finally, he can see how ATAM output affects the quality attributes.

Architectural Decisions

ATAM output (risks, tradeoffs, sensitivities, nonrisks)	AD1	AD2	AD3	AD4	AD5	AD6	AD7	AD8	P	M	PE
R1	√								√		
R2	√				√	√			√	√	
R3		√	√						√	√	
R4			√				√		√	√	
R5								√		√	
T1	√								+	+	-
T2		√							+		-
T3			√						+	+	-
T4				√					+	+	-
T5					√					-	+
S1		√			√	√			√	√	√
S2		√			√	√			√	√	√
S3								√		√	
S4				√			√			√	√
N1	√								√		
N2		√							√		√
N3			√						√	√	√
N4					√	√			√	√	
N5	√									√	
N6					√					√	
	√	√	√	√					P	M	PE
	√		√	√	√	√	√	√	M	**Quality Attributes**	
	√	√	√	√	√				PE		

Figure 6: Architecture Reactive View (ARV) consolidating the disparate tables generated for each scenario into a single artifact. P, M, and PE refer to portability, modifiability and performance respectively.

5 Conclusions

We have used ATAM to evaluate an architecture that is used to develop games which are portable to multiple game engines and modifiable while still playable at acceptable performance. The goal was to find the architectural decisions that contributed to the wanted quality attributes. ATAM managed to reveal and scrutinize these architectural decisions and label them as supportive or undermining. It also managed to identify risk themes that need to be addressed. To conclude we have found ATAM extremely useful in revealing the strengths and weaknesses of the architecture in away that we did not see when using the ad hoc way. It also provided us with a platform from which we can evolve our architecture with confidence knowing exactly what decisions are very sensitive and need to be closely monitored, what decisions need to be remedied, and what decisions are crucial for the success of our architecture.

References

[1] Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L., and Zaremski, A. "Recommended Best Industrial Practice for Software Architecture Evaluation". Technical Report, Software Engineering Institute, CMU/SEI-96-TR-025, ESC-TR-96-025, January 13, 1997.

[2] Adams, R., Ko, A., Nichols, J., Wobbrock, J. "Project Alexandria". April, 2003. http://loki.lokislabs.org/portfolio/alexandria_report.pdf

[3] Adobbati, R., Marshall, A., Scholer, A., Tejada, S., Kaminka, G., Schaffer, S., and Sollitto, C. "Gamebots: A 3D Virtual World Test-Bed For Multi-Agent Research" Proceedings of the International Conference for Autonomous Agents, Workshop on Infrastructure for Agents, MAS, and Scalable MAS, Montreal, Canada, May, 2001.

[4] Aha, D.W. and Molineaux, M. "Integrating learning in interactive gaming simulators." Challenges of Game AI: Proceedings of the AAAI'04 Workshop (Technical Report WS-04-04). San Jose, CA: AAAI Press, 2004.

[5] Bahsoon, R. and Emmerich, W. "Evaluating software architectures: development, stability and evolution." In the Proceedings of ACS/IEEE International Conference on Computer Systems and Applications, Tunis, Tunisia, July 14-18, 2003.

[6] Barbacci, M., Ellison, R., Lattanze, A., Stafford, J., Weinstock, C., and Wood, W. "Quality Attribute Workshops (QAWs), Third Edition", Technical Report, Software Engineering Institute, CMU/SEI-2003-TR-016, ESC-TR-2003-016, August, 2003.

[7] Bass, L., Clements, P., and Kazman, R. "Software Architecture in Practice". Addison Wesley, 2003, ISBN: 0321154959.

[8] BinSubaih A., Maddock, S., and Romano, D., "Game Logic Portability". ACM SIGCHI International Conference on Advances in Computer Entertainment Technology ACE 2005, Computer Games Technology session, Valencia, Spain, June 15-17th, 2005.

[9] Bishop, L., Eberly, D., Whitted, T., Finch, M., and Shantz, M. "Designing a PC Game Engine" IEEE Computer Graphics and Applications, January, 1998.

[10] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. "Patter-Oriented Software Architecture: A System of Patterns". Volume 1, 1996, John Wiley and Sons, ISBN: 0471958697

[11] Chandrasekaran, B., Josephson, J., and Benjamins, V. "What are ontologies and why do we need them?" IEEE Intelligent Systems, Jan/Feb 1999, 14(1), pp. 20-26.

[12] Clements, P. "Active Reviews for Intermediate Designs" (CMU/SEI-2000-TN-009), August, 2000.

[13] Clements, P., Kazman, R., Klein, M. "Evaluating Software Architectures: Methods and Case Studies". Addison Wesley, 2002, ISBN: 020170482x.

[14] Friedman-Hill E. 2003. "JESS in Action", Manning Publications Co, 2003, ISBN 1930110898.

[15] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. "Design Patterns: Elements of Reusable Object-Oriented Software" Addison-Wesley, 1995, ISBN: 0201633612.

[16] Kazman, R., Bass, L., Abowd, G., and Webb, M. "SAAM: A Method for Analyzing the Properties Software Architectures," Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 1994, pp. 81-90.

[17] Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H. Carriere, J., The Architecture Tradeoff Analysis Method, ICECCS '98, Proceedings of Fourth IEEE International Conference on Engineering of Complex Computer Systems, 1998.

[18] McShaffrey, M. "Game Coding: Complete", Paraglyph Press, 2005, ISBN: 1932111913

[19] Ousterhout, J. "Scripting: Higher-Level Programming for the 21st Century", IEEE Computer, 31(3), 1998, pp. 23–30.

[20] Trowbridge, D., Roxburgh, U., Hohpe, G., Manolescu, D., and Nadhan, E. "Integration Patterns: Patterns & Practices", Microsoft Press, 2004, ISBN: 073561850X.

[21] Young, R.M.; Riedl, M.O., Branly, M., Jhala, A., Martin, R.J., and Saretto, C.J. "An architecture for integrating plan-based behavior generation with interactive game environments". *Journal of Game Development* , 1(1), 2004, pp. 51-70.