Generic and Distributed Runtime Environment for Model-driven Game Development

Sebastian Apel and Volkmar Schau
Department of Computer Science, Friedrich Schiller University Jena, Ernst-Abbe-Platz 2, 07743, Jena, Germany

Keywords: Software Architecture, Adaptive, Generic Infrastructure, Game Model, Model-Driven Game Development, Massive Multiplayer Online Games.

Abstract: Massive multiplayer online games are large-scaled distributed systems to handle a huge amount of simultaneous players. Thus, development costs can be enormous. To deal with this, it is necessary to reduce redundant development steps in such distributed systems, e.g. by using code generators and model analysers to build components from already existing knowledge. Such knowledge could be the unique game logic. This paper reports realized approaches to derivate infrastructure from that logic within our middleware. Getting through this is achieved by using an abstract meta model for game design processes, harvest information from those design results and generate infrastructure for communication, controlling and persistance. Finally we evalutes its applicableness and useablity in multiple game projects.

1 INTRODUCTION

In the last decades gaming is dominated by online games – everybody wants to play, sometimes with friends, sometimes against them – furthermore, they want to play by browser, phones or in social networks. Such large-scaled distributed and complex systems could need a high amount of development efforts. Moreover, social, role-playing or shooter games have an intersection of their architectural components. Constructing such Massive Multiplayer Online Games (MMOG) requires an architecture to manage a high number of players, handle their requests and persist their states.

A common MMOG architecture is based on client-server structures (Chen et al., 2013; Bharambe et al., 2006). The server in general contains a continuously simulated game logic, a database to persist game states and a component to handle requests received from clients. The clients on the other side manages subsets of the game state, loaded from the server, can communicate over networks the servers by using a message handler and provides components to handle visual and auditive elements of the game (Chen et al., 2013).

The main difference between concepts for this kind of games can be found in their game logic. This logic describes core elements of a game and the dependencies between them. Furthermore, this is the unique part of a game idea. Each unique game idea has to model and implement the logic as well as construct the infrastructure components around. The question would be about which of these infrastructure components around game logics can be derived and generated by using code generators and a generic runtime environment. Another important question is how to design the architecture to execute such games with generic infrastructure?

The primary objective is to provide a game middleware. The GameEngine, our middleware, has to provide methods to add individual game logics, derives required infrastructural knowledge from them and generates the corresponding components. Using this engine would reduce the amount of development effort and avoids bulky reimplementation of redundant components.

To achieve the goal of generic components, based on a previously modeled game logic, three steps will be necessary and part of our work: (1) an efficient abstract communication layer which avoids calculation overhead mostly triggered by abstraction (Apel et al., 2014), (2) a mechanism to derive information from game logic to generate infrastructure for our game and (3) a meta model for game logics as an entry point for derivation processes and as development support.

This paper will evaluate the question about the useablity and applicableness of this middleware within different kind of game projects. The experi-

Apel, S. and Schau, V.
Generic and Distributed Runtime Environment for Model-driven Game Development.
DOI: 10.5220/0005652306230630
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 623-630
ISBN: 978-989-758-168-7

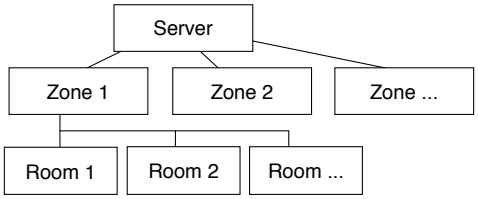

Figure 1: Visualization of a room-based communication structure (gotoAndPlay(), 2011).

mentees are supposed to define their own game idea and going through their model-driven engineering process until they reach their final platform independend game model. The intended game projects are circumscribed to location-based browsergames, a special subtype of MMOGs. Finally each team has to implemented their model and should executed the implementation within the runtime environment of our middleware in addition to some kind of client-side visualization. Furthermore their shouldn't be any technological details like communicating with User Datagram Protocol (UDP) / Transmisson Control Protocol (TCP), serialize data with Extensible Markup Language (XML) / JavaScript Object Notation (JSON) or persist data to MySQL databases.

2 CLASSICAL MMOG DEVELOPMENT

Related research is focusing on architectural questions about how to construct games for massively distributed systems and huge amount of players (Chen et al., 2013; Gregory, 2009; Bharambe et al., 2006). Another part is going through abstracted communication, unfortunately independently from the performance as well as from modelling and game architecture. Combining them into a development process which is dominated by model-driven engineering methodics by using generic approaches is missing.

Taking a look into middleware and tools for MMOG development is dominated by three solutions: (1) SmartFoxServer 2X (gotoAndPlay(),), (2) ES5 Electroserver (Elektrotank,) and (3) Photon Server (Exit Games, a). These Java- and C#-based solutions helps to implement a server-client based game by using TCP or UDP based sockets and a key-value-based proprietary protocol for data (de-) serialization. All products offer a room-based communication infrastructure to manage the distribution of data packets to connected players (gotoAndPlay(), 2011). A room-based infrastructure as shown in Figure 1 could be compared with concepts in instant messengers like Internet Relay Chat (IRC) and ICQ, as well as mes-

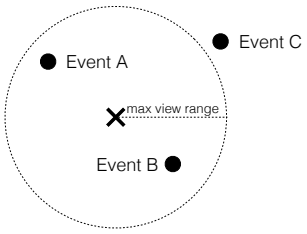

Figure 2: Visualization of a position based communication structure. The centralized position (marked by X) has a specific max view range, each event in this range would be broadcasted to X (in this case event A and B).

sengers based on Extensible Messaging and Presence Protocol (XMPP). Each connected individual can specify a set of rooms to listen for. In case of communication events, the runtime environment tries to place them to the related room and broadcast the contained data to listeners in this room. You can find such solutions in a wide range of games: every time there is a visual separation in the game – for example zone borders, beaming, instantiation or clustering – there is usually a room concept behind the scenes.

The opposite approach of this room based communication would be a location-based mapping of listening individuals as shown in Figure 2. Each individual could have one or more positions and each position will have a specific range to listen to. In case of any event, you will receive this event, if it is released within the range around one of your positions. Only the Photon Server (Exit Games, b) offers such a concept in addition to room-based listenings. Apart from that, all middleware offer client implementations in common technologies like HTML5 / JavaScript, Flash / ActionScript, Unity, iOS and Android.

To add a game idea to these middleware, all solutions offers so called extension points. This extension point within this monolithic architecture offers capabilities to interact with the communication infrastructure by using a specified Application Programming Interface (API). In case of any game logic the developer has to implement their own controller layer to handle messages between middleware and game logic.

3 ABSTRACT META MODEL FOR GAME LOGICS

The objective of this paper is to identifying methods for game logic analyses. These methods will be combined in our GameEngine to support model-driven game development processes. First of all, we need to know detailed information about the game logic itself. Defining an abstract model can solve this. This

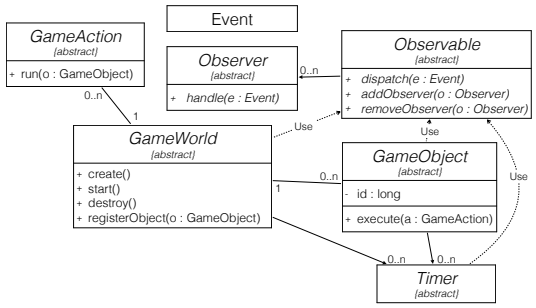

Figure 3: Class diagramm of a basic meta model to use in game logic implementations.

model defines entry points which can be used by an abstract component for further analyses. Furthermore this model has to be the intersection, which could be found in each game logic. To reengineer such a meta model, a closer look on fundamental principles of game modeling was necessary.

"A game is a type of play activity, conducted in the context of a pretended reality, in which the participant(s) try to achieve at least one arbitrary, nontrivial goal by acting in accordance with rules." (Adams, 2010) Pretend some kind of new reality, define a set of rules and nontrivial goals (Adams, 2010; Rollings and Morris, 2004). This non-existing, virtual "pretended reality" is our base element within the abstract meta model. The GameEngine will call this element "game world". The element "goal" is a special rule, a rule with a positive termination condition. Finally the "rule" should be added indirectly to handle different types like: (1) semiotics of the game to describe meanings and relationships for various symbols, (2) gameplay to describe challenges and actions, (3) sequence of play for progression of activities, (4) the already pointed out goal of the game, (5) some termination conditions and (6) meta rules about these rules (Adams, 2010).

Rules, which describe "something" in this game world, are semiotics of the game, especially their symbols (Adams, 2010). The GameEngine will call this symbols "game objects". Whenever a rule describes something like an avatar for the player or an item like a sword, which could be equipped by this avatar, then this rule has to be implemented as a specialization of an abstract game object. In addition to the semiotics of games, there is a need to realize the other types of rules. One more element of the meta model is our set of actions as a part of the gameplay rules. Each game object can trigger a set of possible actions, hereinafter called "game action". Combining **game world**, **game object** and **game action** will create the first version of the abstract game meta model. Figure 3 shows this abstract game meta model and extends this by using an observer pattern

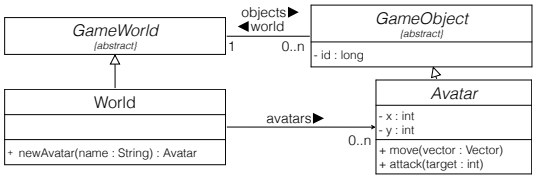

Figure 4: Example model for a simple 2D game. The player can control the avatar and attack other players.

and timer component. The observer pattern (Gamma et al., 1994) helps to manage state changes as well as special events. The timer is used to simulated the model independently from user interactions.

4 DERIVATE INFRASTRUCTURE FROM GAME LOGIC

In case of using a game logic which is based on the previously defined meta model, the second step would be the identification of significant elements to derive an infrastructure. The simplified game logic shown in Figure 4 will be used to describe requirements and how to get to communication components.

4.1 Identify Communication Protocol

Figure 4 shows a game, where a player can control his avatar by moving in two dimensions, for example by pressing the arrow keys on his keyboard. To join this game, the player needs to create a new avatar. To achieve victory points, the player need to attack other avatars. In case of manually creating an infrastructure-workflow as shown in Figure 5 based on common MMOG architectures, the following steps have to be done by developers:

1. Creating request objects to name the special request and define their individual parameters
2. Creating a handling description which will check parameters, map them to match game logic restrictions, call the proper methods and creates a response object
3. Creating a request receiver process and a mechanism to identify the requests and assign them to the correct handling description and returns the results of this handling description
4. Creating response objects with parameters which contains information about the result of a request

These steps can be used as an abstract generalization and can be found in many implementations of request-response-based interfaces. The example

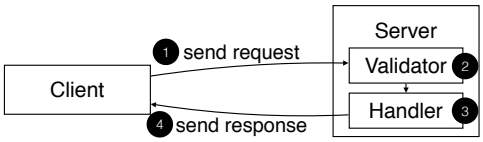

Figure 5: Abstract communication flow for message handling.

model in Figure 4 requires three requests, three handling descriptions and three responses to handle new avatar, move and attack. In detail, the request to handle a new avatar should contain a parameter name, the request to handle moving should contain the avatar id as well as the corresponding vector whereas finally the request to handle the attack should contain the avatar id as well as the target id. By using this example two types of requests can be found: (1) requests with context (move, attack) and (2) requests without context (new avatar).

To derive requests from our example in Figure 4, we search for method signatures within the game world to create requests without specific object context as well as searching for methods within game objects for requests with specific object context. While requests without context only use the parameters of the signature, requests with context have to extend this parameter list by using an identifier for this corresponding game object. Additionally, all related responses can be derived directly by using return values of currently analysed method signatures.

Continuing this way it is possible to derivate handling descriptions. Using parameters and return values to ensure the correctness of request parameters and match them to the game logic by mapping data type differences between request and method could do this. Mapping of data types should be treatable, because previously generated requests. Finally, the call to the proper game logic method: in case of contextual calls, the handling description needs to use an access point to find the game object by using the id. In case of a request without context it would be enough to use the instance of the world we are currently connected and call the matching method. The result of this contextual and context free call can be used to create the response and should be returned.

To create a request receiver process and a mechanism to identify the requests, it is important to collect the generated elements. Because generating a tuple of a request, response and handling description for each necessary method in the example game logic, the receiver process should know the correct handling description for each request and the possible responses. Figure 6 shows how to derive request, response and handler in case of the new avatar method.

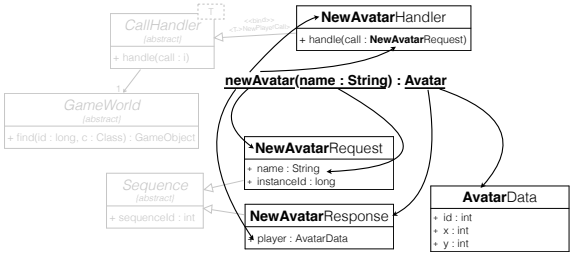

Figure 6: Model based derivation of communication related elements by using thew new avatar method from Figure 4.

4.2 Identify Data Representation

In addition to identify requests, responses and handlers, it is possible to use the game logic to generate a generic data representation. As shown above, we can derive all necessary communication elements. To get a data stream, ready for transmission, we need a method to serialize those elements.

One solution would be to use so called serializers. Most of them produce XML or JSON based on previously annotated entities. In case of implementations based on Java, those tools make heavy usage of less performant reflection API (Oracle,) calls. Using this API help to interact with entities on abstract levels instead of native method calls. Testing the performance loss, in case of using the reflection API, can be done by calling a method multiple times natively compared to calls done with the reflection API, measuring execution time and divide this by the number of iterations. This test will show something around two, up to three, times more time per reflection based method call. Those reflection API calls are used by serialization tools for attribute identification, reading meta information, calling getters and setters and creating new instances of data entities. Taking a plain Java serialization, as reference, shows a significant raise of serialization time in case of popular XML and JSON tools as shown in Figure 7. Figure 7 shows simple entities which encapsulates basic variable types as well as complex entities with combinations of them. Each entity is serialized multiple times by using the same previously allocated space to avoid measurement errors based on allocation processes. To speed up this serialization process it is necessary to avoid reflection based calls as much as possible. Additionally reducing string based operations would help to boost up performance.

The GameEngine acts as a fully generative execution environment for game logics. In case of abstracting communication for model-driven development and hiding infrastructure it would be possible to use more proprietary protocols. This enables the GameEngine to use meta information free data repre-

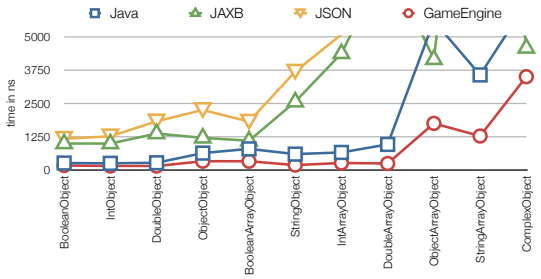

Figure 7: Serialization time of different types of entities with Java, JAXB for XML, JSON and GameEngine.

Figure 8: Data transformation for entity with two integer variables. Header contains a numeric namespace and a numeric identifier for SomeObject. Attribute values will be added after header in alphabetic order.

sentation. Each data stream starts with a little header to identify the represented communication element by using an numeric identifier, followed by a alphabetical order list of attribute values. Value will be encoded as they are, e.g. an attribute with 32 bit integer type will be encoded as a big endian sorted 32 bit integer. Figure 8 shows an example how to transform a simple entity into data streams based on this idea. To get a more save data handling a little checksum is embedded in some data types. Figure 8 contains an additional checksum integers and a checksum for object ids.

To avoid reflection calls the GameEngine uses code generators to create native serialization processes for each previously identified communication entity. Each process is a list of binary operations to extract data from streams to entity instances or to extract data from entity instances to the data streams by using native getter and setter calls as well as binary operations for data transformation.

As shown in Figure 8 the runtime generative code for mappings between stream and entity to avoid reflection based calls is faster than JAXB, JSON or Java itself. Compairing complex data entities with fixed and floating point numbers as well as booleans, strings and sub entities result in 80 times faster transformations. In general it is two times faster than native serialization in Java just by using code generators, binary based data representation and communication objects extracted from specific domain models without losing abstraction level while programming. In addition Figure 9 shows exactly the same entities used in deserialization processes and again with faster runtime than XML and JSON based solutions.

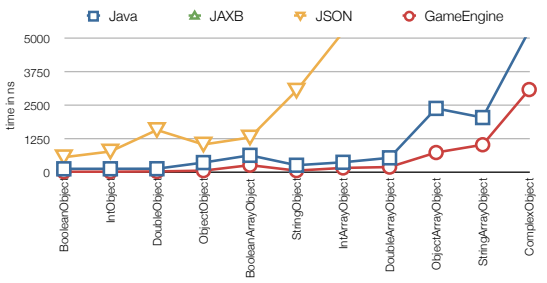

Figure 9: Deserialization time of different types of entities with Java, JAXB for XML, JSON and GameEngine. The runtime of of JAXB is 100 times slower than JSON and not visible in this scale.

This method for entity transformation into data streams could not be used in common development processes easily. Required dependencies between client and server implementation would result in a high failure rate. However, approaches based on model-driven development can be used to embed much more performant ideas and could avoid those failures by using generated elements in development processes.

5 GAMEENGINE ARCHITECTURE

To run a game logic based on this idea the GameEngine uses an architecture as shown in Figure 10 based on a reference architecture for a MMOG (Chen et al., 2013). Our architecture contains already named components like game logic, game state and message handling, but separates a controller from message handling. The message handler manages connections and (de-) serialization features while the controller will handle requests by using the handling descriptions. This modification is based on a Model-View-Controller pattern to keep a cleaner separation of components (Buschmann et al., 1996). The presented Figure shows the architecture for one single game instance. The GameEngine itself comes as an execution environment as shown in Figure 11 written in Java by using well known specifications from the Java Enterprise Edition (EE). The reason why Java EE was chosen is because of its structured components to implement services, business logic and additional interfaces. Using this allows game developers to richly extend their games by using standardized concepts to create additional services around their game logic. Furthermore the game engine can offer optional services to enable authentication, payment, statistical and community services.

The main elements of our middleware are infras-

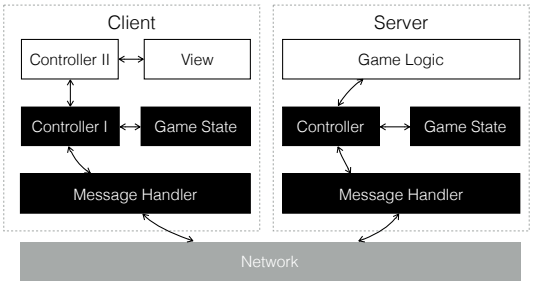

Figure 10: Architecture for a single game instance. Elements marked black could be generated by using the game logic (Chen et al., 2013).

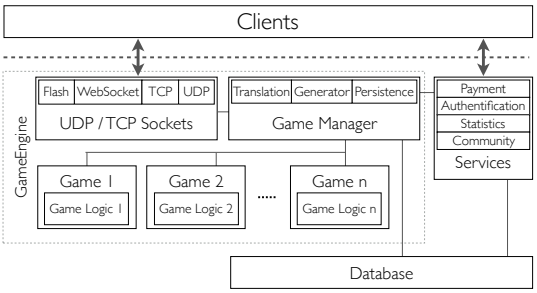

Figure 11: Global GameEngine view; our middleware can handle multiple game logics at once, can offer services and handles public client connections.

tructure, generator, translator, meta model and persistence to realize the described idea as shown in Figure 11. The infrastructure can be compared to common socket implementations, which can be used in Flash, WebSocket or binary mode. The mode for running a connection is switched dynamically by detecting client specific characteristics.

The game manager handles game logics and offers translator, generator and persistence features. In case of adding new game logics, the manager will create the corresponding game instance, analyses the new game logic, creates the new public communication port and initialize the logic itself. Analyses and derivations will be done directly after the start process. Finally, logic and generated elements are prepared, available and ready for use.

The translator is an internal component, used as a runtime service or as a deployment service. In case of JavaScript the translator can be used to dynamically include the current client code version. Alternatively, this component can also be used as a service while deploying a new client version to embed the translated code fragments directly.

6 EVALUATING THIS STRATEGY

The described concepts will work in theory as shown above when using a compact game concept. Next step would be to go through real scenarios, e.g. game projects. These projects starts with a computation-independent model (CIM), followed by a platform-independent model (PIM) and finally moved to a platform specific model (PSM) for a specific game logic (Pastor et al., 2008). While moving through a PSM the created game logic should extended by using the provided meta model for game development as presented above. After all, adding the implementation of the game logic and some client logic for visualization and the game should be done. Initially we evaluate, if this concept is applicable and useable as well as whats about the development savings.

6.1 Viechers

One of the first games created using the mentioned GameEngine is Viechers, which is an evolution based open world strategy game. The player controls a horde of entities to collect resources, to breed his critters, to evolve them and building a civilization by placing constructions. The game idea of Viechers was created in an early state of our middleware. The logical core of this game is an independent game domain model, which describes all entities and their relations, e.g. the “viechers”, rocks and trees. This logic is based on a game world, game objects and a set of game actions for each object, e.g. breeding, harvesting, feeding and sleeping. Because of the early development state of the GameEngine, request and response objects as well as the handling descriptions were created manually. The main goal of this project was a proof of concept to validate the theoretical base in a real game project and evaluate requirements to have an executable example. Furthermore this game was released to public to test the GameEngine performances in real-world conditions. The client is written in ActionScript and the client-side communication layer is translated from server code by using our proprietary code generators as described above. Because of this finallized, complex MMOG and the manually written components we could measure, which elements would have the most effort. The analysed costs were separated into game logic, request/responses, handling descriptions and infrastructure. Figure 12 shows a high load for infrastructure, requests/responses and handling descriptions. The goal would be to avoid this development parts in future.

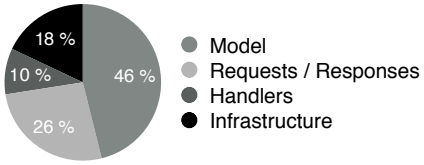

Figure 12: Measurements to show how many line of codes each part of Viechers uses. There is a overall number of about 100kloc (without implementations for visualization).

6.2 Stickman Project

Stickman, as another game, was the first one using the game engine as intended. This game is about a simple stickman running in a 2D adventure to fight against other players. The goal of this project was to show that the GameEngine can handle game logic implementations based on the meta model without additional development efforts for infrastructure and communication. However, this is an ActionScript based game, but in this case the client code is translated from previously derived communication objects. In relation to Figure 12, the infrastructure is part of our middleware. Handlers, request and responses are derivated from the Stickman game logic. Analysing client code shows savings of 75 % comparing visual and infrastructural implementations. The server on the other hand handles 11 different types of requests, one of them is implemented manually. Each request has its own request data object and all requests together utilizes just five different response objects. One of those response object is generated from server code, the other one are implemented as simple data objects with approx. three attributes. Getting objective measurements requires an additional implementation of this project with fully manually written communication and controller layer, which is currently not done. However, the project itself shows the applicableness and a working game idea without any kind of communication handling – just object oriented handling of entities within the Stickman game logic.

6.3 Independent Game Projects

Next step is to involve independent developers for evaluating the middleware within their ideas. In this case four student groups, approx. three experimentees per group, create and implemente their own game by using the GameEngine: Prevolution, Terra Life Evolution, HackNSlay Adventure and Kokusnussbikini. All projects had to follow a real development process starting with preproduction phase, going through production and alpha phase and end up with beta phase (Claypool and Lindeman, 2008). Teams had to start with their idea, without any knowledge about our middleware, as well as creating their game related sketches. This included requirement analysis, creating of use cases as well as first drafts of their game logic with UML tools for class diagrams. Furthermore, these steps are handled without knowledge about our middleware. After that, the implementation phase started. Teams got an introduction to our middleware and get their training about how to use the API in combination with their created game logics.

Prevolution is a prequel for Viechers by using the current version of the GameEngine. The player controls single-cell organisms and tries to evolve them by swimming around in the tide pool. The client is written in Java, so translation is unnecessary. This project helps to test and evaluate a pure Java implementation for server and client as well.

Terra Life Evolution is about a classical role-playing concept with modifications to handle the skill development. Central scene of this game project is a world separated in islands, which again are separated into several areas. The player starts in a constant area, with matching enemies to manage their first steps, perform quests and improve experience to get skills.

The **Hack'N'Slay Adventure** is a round-based concept where players have to manage defending a centralized object and protect this one against onrushing enemy troops. The game starts with a lobby, where players were put into teams. Each team starts with an instantiated map to defend the centralized object. After ten rounds of onrushing enemy troops the game is over, depending on survivors the player teams wins or loses the match.

The last one, **Kokosnussbikini**, is based on an open-world-survival idea. This game describes a generic world for players to move, defend, collect resources and place objects. The client uses an isometric visualization to show player and landscape.

These four projects have shown that it is possible to develop browserbased MMOGs by following classical development processes and avoiding development overhead for infrastructure and communication by using our meta model for their game logic. In relation to Figure 12, none of these projects have any kind of socket implementation, neither any line of code to define request or response objects or to handle them. Also there is no additional effort to handle different client and server languages. Furthermore, these projects gave a closer look at the following problems:

- Challenging differences between coding languages, for example the syntax for getter and setter. Currently the GameEngine matches them as good as possible by using the syntax from Java copied to other languages. The opposite would be to use the client side language syntax for them,

which could lead to problems in finding the correct spelling.

- Another problem is related to missing mechanism for synchronous calls on the GameEngine in ECMA based scripting languages (for example JavaScript or ActionScript). To manage this language specific behavior, the GameEngine generates additional parameters in method signatures for success and failure callbacks.
- JavaScript does not distinguish between byte, short, integer, long, float or double. The GameEngine middleware tries to address these restrictions, however, it may lead to inaccuracies. In normal cases, this will not affect the development.

7 CONCLUSION AND FUTURE WORK

This paper describes the idea of extracting infrastructural components from implemented game logics by using model-driven engineering methods and to evaluate useablity and applicableness in realistic conditions. Development can be focused on modeling the game logic which means handling of symbols and relationships between them. Furthermore, it is possible to reduce infrastructural implementation efforts dramatically, which reduces testing and raises overall stability by using an adaptive and generic execution environment.

Using model-driven engineering in a highly abstracted framework in addition to code generators, enables usage of proprietary mechanisms hidden inside a middleware. They could boost internal critical components without any need to get a developer in touch with it. They just use their own modeled and implemented entities within server and client.

So far there is a working middleware to develop games based on this idea as shown in independent game projects, mostly location-based Browser-MMOGs. In future work this project aims at comparing game logic implementations. Using at least one of them in combination with a manually constructed communication layer would give a brief insight about the percentage of savings.

ACKNOWLEDGEMENTS

We would like to thank all members at the Department of Computer Science here at FSU Jena – especially Wilhelm Rossak. We would also like to extend our gratitude to Bernd Weigel and all students here at FSU Jena about their contributions in different kinds of game projects as well as the GameEngine itself.

REFERENCES

Adams, E. (2010). *Fundamentals of Game Design*. New Riders, Berkeley, 2nd edition.

Apel, S., Schau, V., and Rossak, W. (2014). Erprobung von generischen Kommunikations- und Verwaltungsinfrastrukturen in modelgetriebenen Entwicklungsprozessen bei ortsbasierten MMOGs. In *11. GI/KuVS-Fachgespräch "Ortsbezogene Anwendungen und Dienste"*, volume 11. in press.

Bharambe, A. R., Pang, J., and Seshan, S. (2006). Colyseus: A distributed architecture for interactive multiplayer games. In *NSDI '06: 3rd Symposium on Network Design and Implementation*, San Jose, California, USA.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *A System of Patterns, Pattern - Oriented Software Architecture*. Wiley, 1 edition.

Chen, Q., Galiullin, A., and Woo, J. (2013). A reference architecture for multiplayer online games.

Claypool, M. and Lindeman, R. W. (2008). Game development timeline. Internet, http://web.cs.wpi.edu/ imgd1001/a08/slides/imgd1001_04_GameDevTimeline.pdf, visited 2015-10-21.

Elektrotank. http://www.elektrotank.com, visited 2014-09-12.

Exit Games. https://www.photonengine.com, visited 2015-10-21.

Exit Games. Mmo concept. Internet, http://doc.photonengine.com/en/onpremise/current/reference/mmo/mmo-concept, visited 2015-10-21.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition.

gotoAndPlay(). http://www.smartfoxserver.com, visited 2015-10-21.

gotoAndPlay() (2011). *SmartFoxServer 2X Zones and Rooms Architecture*. Internet, http://docs2x.smartfoxserver.com/Overview/zones-room-architecture, visited 2015-10-21.

Gregory, J. (2009). *Game Engine Architecture*. A K Peters/CRC Press, Boca Raton.

Oracle. Trail: The reflection api. https://docs.oracle.com/javase/tutorial/reflect/, visited 2015-10-21.

Pastor, O., España, S., Panach, J. I., and Aquino, N. (2008). Model-driven development. *Informatik Spektrum*, 31(5):394–407.

Rollings, A. and Morris, D. (2004). *Game Architecture and Design: A New Edition*. New Riders, Indianapolis.