

Fast and Safe Prototyping of Game Objects with Dependency Injection

Erick B. Passos
Media Lab - UFF

Jonhny Wesley S. Sousa
LCD - UFCG

Giancarlo Nascimento
Media Lab - UFF

Esteban Walter Gonzales Clua
Media Lab - UFF

Lauro Kozovits
UERJ

Abstract

Most game engines are based on game objects inheritance and/or componentization of behaviors. While this approach enables a clear visualization of the system architecture, good code reuse and fast prototyping, it brings some issues, mostly related to the high dependency between game objects/components instances. This dependency often leads to static casts and null pointer references that are difficult to debug. In this paper we propose the use of the Dependency Injection design pattern to safely initialize game objects and alleviate the role of the programmer in the handling of these issues both during prototyping and production phases. Since these dependencies are attributes in game objects and the injection occurs only at the initialization pass, there is no performance penalty at the game loop.

Keywords:: game engine architecture, dependency injection, object composition

Author's Contact:

{epassos,esteban}@ic.uff.br
jonhny@lsd.ufcg.edu.br
giancarlotaveira@gmail.com
lauro@jogos.etc.br

1 Introduction

Few domains in computer science map to a programming paradigm so directly as computer games to object orientation. A *Game* class matches the concept of a virtual world where several different instances of a *GameObject* class reside. A *GameObject* instance can be anything such as the player character, a house or an invisible trigger object. The game execution usually consists of a loop inside the *Game* class with three main goals:

1. Collect user and network input;
2. Update each *GameObject* instance based on input and/or physics simulation, animation and Artificial Intelligence step;
3. Draw visible game objects on the output device.

There are alternative forms of this loop proposed to better exploit parallelism [de Moraes Zamith et al. 2007], but we consider that this basic model serves well to express the purposes of our work.

To represent different types of simulated objects, the programmer usually creates subclasses of *GameObject*, each one specifying new and more specialized content and behavior. The problems with the inheritance approach are well known. A good example of this is when two objects from different hierarchies sometimes have common functionality and characteristics, leading to redundancies in code. This is a common issue in object oriented software design and replacing inheritance by composition in game engines is already recognized as a good practice [Folmer 2007; Stoy 2006; Ponder 2004; Billas 2002].

In its componentized alternative, the *GameObject* class holds only common attributes that all game objects should have such as name, position and orientation. Most important, however, is that it acts as a container for reusable components. Each class extending *AbstractComponent* represents a different aspect/behavior of a *GameObject* such as physics, AI or health, depending on the needs of the game object being (now) composed. These components should be highly flexible and easier to maintain while also

maximize code reuse as many of them are useful in several different game genres.

Both approaches have a common issue, related to high coupling between components (or game objects). For instance, an AI component commonly depends on the existence of a health component to decide actions to take and also on a physics component to apply movements to. These dependencies are normally resolved directly by the programmer, as shown in the following Java code, part of an *AComponent* class, adapted from the original C++ example [Stoy 2006]:

```
1 public void update(float interpolation) {
2     final GameObject o = getOwner();
3     Health h = (Health) o.getComponent("health");
4     if (h != null) {
5         // take AI actions based on health
6     }
7 }
```

Code 1: Traditional dependency handling

It's easy to see that this implementation has an implicit dependency on the existence of a *Health* component (line 3), which is expected to be registered at the same game object under the label "health". If every game object containing an instance of this *AComponent* class is properly initialized with an accompanying *Health* component, everything will work as planned by the programmer. Apparently there is nothing wrong with this code, but a closer inspection shows us that:

- While necessary for solving the implicit dependency, lines 2-4 have nothing to do with the expected game logic of an AI update, being just boilerplate code;
- The explicit cast in line 3, or the absence of strong typing in the case of some script languages, is a common source of runtime problems in dependent component/objects;
- If no *Health* component is initialized for this particular game object, no proper AI action (code inside if clause) will ever be taken, making it harder to debug while doing level design (unless the component logs the unfound dependencies, something the programmer would have to code directly).

An equivalent code in UnrealScript [EpicGames 1998] would be even more difficult to debug because the language hides all null pointer references by making them equivalent to a void execution line. This is such an issue that Tim Sweeney have recently said that around fifty percent (50%) of the bugs found in Unreal were related to this kind of dependencies and the lack of stronger typing in its programming languages [Sweeney 2006]. He also points out that a typical game object update usually touches five to ten other objects, which shows how relevant and frequent the problem is.

In this paper we propose the use of the Dependency Injection design pattern [Jin 2007; Fowler 2004] to solve part of this problem by freeing the programmer from the responsibility of manually checking for these dependencies. As will be shown in the next sections, our framework takes care of the safe initialization of game objects and components, which can be coded in a much cleaner and maintainable fashion. The rest of the paper is organized as follows: section 2 discusses related work, section 3 presents the concepts and design patterns implemented by the GCore framework while section 4 explains the use of Dependency Injection and the advantages of our framework over previous research. Finally, section 5 concludes the paper and outlines future work.

2 Related Work

Successful Game engines from the industry are strongly based on the game object inheritance model such as CryEngine [CryTek 2008] with its entities and entity-items being equivalent to game objects and components respectively. The same architecture is found in other commercially available engines such as UnrealEngine [EpicGames 1998] and Torque [GarageGames]. At the same time there are more consistent componentized architectures [UnityTechnologies 2008; Spinor; 3DVia; Billas 2002]. The problems related to high dependency between objects are a common issue of these tools, making them potential targets to our proposal.

In a talk presented by Tim Sweeney [Sweeney 2006], he exposed in detail two problems with current game programming languages and tools: poor concurrency handling and weak typing. He proposes some features that a new programming language should have that could solve most of the runtime bugs programmers deal with when implementing game object scripts. While his arguments are somehow similar to ours, he proposes the creation of new languages with the suggested features, which is not a simple task. In this paper we are proposing the use of a currently available technique that can be adapted in most existing tools. Our Dependency Injection implementation is based on Java reflection features, being trivial to port to C# based platforms such as XNA [Microsoft]. C++ and some script languages doesn't have the exact reflection features used by our framework but it is possible to implement the same idea with some workarounds [Pocomatic 2007].

Dungeon Siege was one of the first games to include a fully componentized game object system. In two different years at the Game Developers Conference [Billas 2002; Billas 2003], Scott Billas showed how this architecture and some other features helped the development of the game and its "continuous world". In a more recent talk [Billas 2007], he exposed ideas on how to improve a game production pipeline, several of them being related to sanity checks during game objects/components initialization such as attribute requirements and dependencies. He proposes that these assertions and error messages should be implemented directly by the component programmer/engineer on the behalf of the level designer. We recognize that these ideas are very important indeed, but also aim to provide tools to automatically perform these sanity checks and also dependency solving/injection, this time on the behalf of the programmer/engineer and consequently improving the whole production pipeline.

Haller et. al. [Haller et al. 2002] proposes a new architecture for game objects and components using communication slots and a message manager to remove the strongly typed dependencies between them. The solution enables easy composition and connection of components but requires a strong commitment to a more complex architecture. With our approach the programmer doesn't have to learn any new language or communication architecture, making it more suitable to fast prototyping.

The Unity3D game engine [UnityTechnologies 2008] is a relatively recent product that has gained attention from developers given its well designed game object component system and scene editor, which uses a visual approach to composition. In its latest version (2.1, released in late July, 2008), a simplified form of dependency sanity checking is provided for the script programmers, who can specify that a component has a dependency on the existence of another, which is checked at runtime and also inside the scene editor. However, although automatically instantiated by the scene editor, this instance is not automatically injected in the dependent component, leaving this responsibility to the programmer, who still has to explicitly call a `GetComponent(Type)` method to obtain the reference. Our system does both the sanity checking and the automatic injection (reference solving) of dependent components.

To our knowledge, all previous research and/or products only go so far in the development of game object component systems. We propose the full adoption of Dependency Injection to handle the coupling of components in a game engine. In the following sections, our framework, named GCore, will have its architecture explained together with the application of Dependency Injection in game objects composition.

3 GCore Architecture

GCore, abbreviation for Game Core, is a data-driven game framework aimed at high productivity that abstracts the use of JMonkeyEngine [JMonkeyEngine], a Java scene-graph engine implemented over hardware-accelerated OpenGL for graphics and OpenAL for audio. GCore also includes add-ons such as a physics engine [JMEPhysics] and a network subsystem [Imagination] to deliver an extensible and easy to use tool. Since its core concepts are feasible to implement in other platforms and languages such as C++ or C#, we focus our attention on its data-driven approach and Dependency Injection features, which enable game designers, level designers, programmers and artists to cooperate seamlessly in a game production pipeline. The role of the programmer is to code reusable components implementing the game design ideas while the level designer integrate these components with the assets created by artists. In this paper we show GCore tools and techniques to assist both programmers and level designers.

3.1 Main Concepts

From a software engineering perspective, GCore defines four main concepts/classes to represent a game: GameManager, GameState, GameObject and AbstractComponent. GameManager is a Facade [Gamma et al. 1995] for the whole system while a GameState is similar to a use-case scenario, each one isolating a game scene (or other user interaction concept) such as the 3D playing environment, a menu or a heads up display (HUD). During execution, the player alternates through these scenarios as the game switches between the different instances of GameState. A runnable GameState is defined with an unique name and containing a collection of GameObject instances, each one composed by a collection of AbstractComponent implementations. In Figure 1 it is possible to see a simple class diagram for GCore architecture while in Figure 2 one can see the sequence diagram that explains the game loop concept used in GCore by specifying the order of method calls at each discrete step.

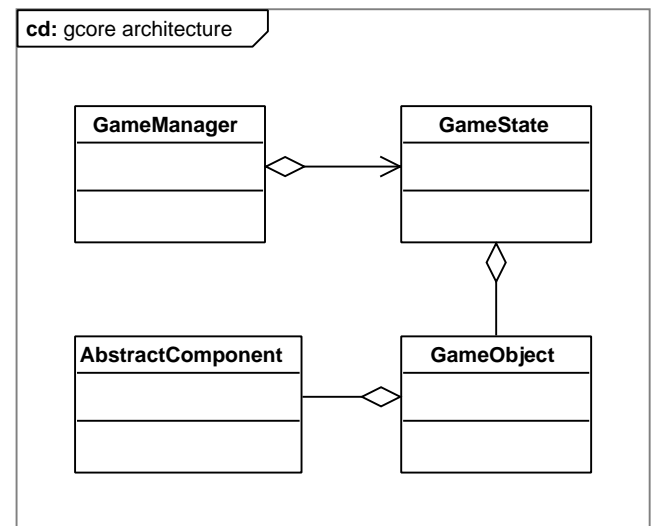


Figure 1: GCore componentized architecture

As can be seen from the diagrams, each component is updated at every frame. There is no render method in the AbstractComponent class because most of them do not represent graphical properties of the object. Instead, the GameObject class keeps a scene-graph node where any graphical component can attach a drawable geometry if needed. This node is rendered at the end of each frame step which leads to the rendering of the geometries of graphical components attached to the game object.

3.2 Data-driven Composition of Game Objects

The main reason for GCore's productivity is the possibility of creating a game entirely in declarative form. An example of compo-

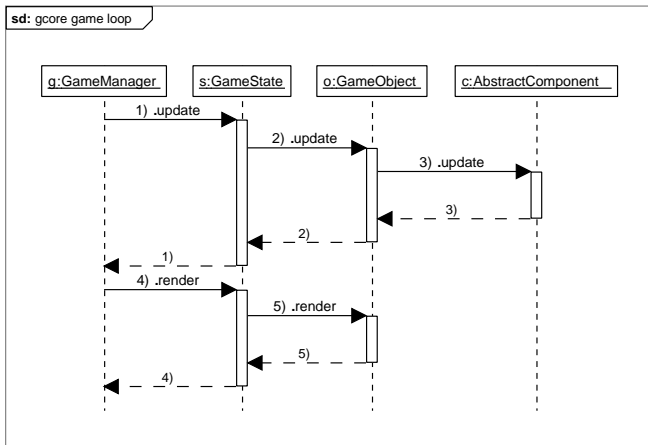


Figure 2: GCore game loop

sition is shown in Figure 3, an object diagram where a GameState is composed of two GameObject instances, each one with different components representing their characteristics and behaviors.

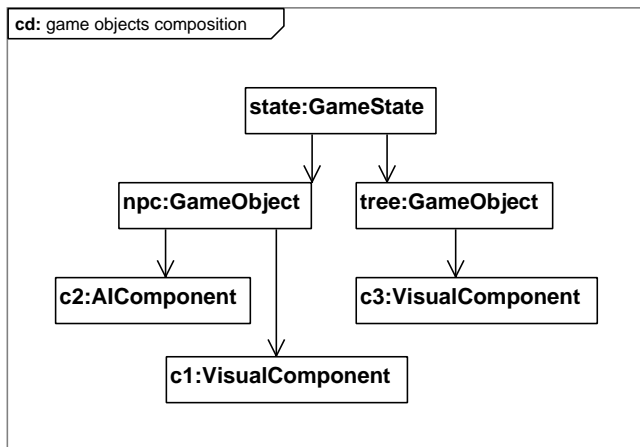


Figure 3: Composition example

In the above example, there are two active objects in the game state named "npc" and "tree" respectively. The "npc" object is composed of an instance of VisualComponent carrying its graphical representation and an AIComponent, responsible for controlling its actions during game execution. Notice that since the update method is called at every frame step, both components are updated. In the AIComponent this method contains the actual implementation of the AI step, while in the VisualComponent this method is empty. The other game object, named "tree", is static and consists only of a graphical geometry, represented by a single VisualComponent.

Game states and their companion game objects are all stored in XML files, which are very easy to maintain and also suitable for an integration tool such as a level editor. In Code 2 one can see the main XML configuration file of a simple game. The root element of the configuration is the "game" element, which has two attributes: a name and the first game state to be loaded. The "game" element must be composed of type declarations and game state compositions, being possible to keep them in as many separate files as desired, feature exemplified by the use of the "include" tag.

Type declarations provide for a way to compose reusable types that can be further specialized and instantiated into game objects inside the game states. Types are composed of components which can have their attributes values also specified inside the XML files. In Code 3 one can see a type declaration showing the concepts of composition, inheritance and attribute specification. The first type, named "basic", defines that all derived types and objects will have a VisualComponent attached. The second one, named "npc", ex-

```

<game name="example" init="menu">
  <!-- type definitions -->
  <include file="types.xml" />

  <!-- game states -->
  <include file="menu.xml" />
  <include file="farm.xml" />
</game>
  
```

Code 2: Main configuration file (game.xml)

emplifies inheritance by extending "basic", from which it brings the VisualComponent and also attaches a new AIComponent. The "tree" type shows the specification capabilities of GCore by inheriting from "basic" and modifying an attribute in the VisualComponent, in this case defining an external 3D model to be loaded as geometry for any "tree" typed game object.

```

<type name="basic">
  <component class="VisualComponent" />
</type>

<type name="npc-type" extends="basic">
  <component class="AIComponent" />
</type>

<type name="tree-type" extends="basic">
  <component class="VisualComponent">
    <model value="tree.3ds" />
  </component>
</type>
  
```

Code 3: Sample type declarations (types.xml)

Game states are composed of game objects, which can inherit from pre-defined types and specify or include any component as needed. One can even define objects without a supertype, but this practice minimizes reuse and is not recommended. In Code 4 one can see the XML for the game state shown in Figure 3 in Page 3. The "npc" game object inherits from the previously declared "npc-type" and specifies a 3D model for its VisualComponent. The "tree" object specifies a new position vector for its VisualComponent. One can easily notice the flexibility of this data-driven approach for game objects composition and it is even possible to have multiple named components of the same class in the same type. The classes GameObject and AbstractComponent also implement the Composite design pattern [Gamma et al. 1995], being possible to have chain of nested components if needed.

```

<gamestate name="farm">
  <!-- game object1: npc -->
  <object name="npc" type="npc-type">
    <component class="VisualComponent">
      <model value="zombie.3ds" />
    </component>
  </object>

  <!-- game object2: tree -->
  <object name="tree" type="tree-type">
    <component class="VisualComponent">
      <position x="10" z="15" />
    </component>
  </object>
</gamestate>
  
```

Code 4: Game state and objects composition (farm.xml)

3.3 XML parsing and game execution

From the last section one can see that each game is completely specified by a meta-data configuration stored in XML files. This XML specification is parsed during initialization and loaded into a set of configuration objects. This light-weight meta-data structure is used at runtime to instantiate game states and game objects as needed. GCore is capable of parsing attribute values of all Java primitive types and also 3-valued vectors, quaternions and assets (files) such as textures, models and audio clips. It is also possible to create your own parser for any user-defined structured type (class) by extending the PropertyParser utility class.

There has to be at least one default game state in a game, which will be the first (or only) to be loaded. Since a game is composed by a collection of such game states, there should be a way to instantiate, destruct or switch between them at runtime. To provide for an elegant implementation of these features, the GameManager class implements the Mediator design pattern [Gamma et al. 1995] with public methods to (re)activate, pause or destroy any declared game state by its name. The destroying of previously enabled game states is optional (prior the activation of another one) since more than one can coexist in memory at the same time.

The GameManager also take care of the correct initialization of all game objects and its components. The Builder design pattern [Gamma et al. 1995] was applied in the initialization code for game states, objects and components, taking care of attribute and dependency injection. However, for the sake of readability, in the next section we will show a simplified version of this process instead. The reason is that we will focus on the explanation of the dependency injection usage, its advantages and implementation details.

4 Dependency Injection in GCore

Dependency Injection, sometimes referred to as inversion of control, is a design pattern that provides a flexible way to indirectly assembly dependent software components together [Jin 2007; Fowler 2004]. In a complex componentized system, such as a game, it is common to find classes that depend on others to perform their tasks. The programmer usually implements this dependency as an attribute declared with its type being the target class. Dependency solving normally occurs at runtime by getting an instance using an available pull-like API.

With Dependency Injection the underlying framework is responsible for automatically look for this instance and setting it under the dependent component. There are two gains associated with the use of this pattern: smaller and cleaner code at the components, since it's not necessary to manually look for the dependency; and safer initialization, because it's a responsibility of the framework to check for the existence of the dependencies, leading to less runtime errors.

To implement this technique we make use of the runtime reflection features of the Java language, especially annotations, which are a special kind of meta-data available both at compile and runtime. We created a custom annotation type, named @Inject, to be used to mark dependencies between components in the source code. These dependencies will be solved at loading time by the game object initializer every time an attribute identified with @Inject is found. In the following section we explain two examples of Dependency Injection in our framework: the composition of a player game object showing the safe initialization of dependent components in the first example and the fast prototyping of new game mechanics in the second one.

4.1 Example 1: Safe Initialization of Components

In GCore, all direct dependencies between components can be automatically solved by the framework. First lets consider how very simple game object features can be implemented as reusable components: external 3D model loading, player input and a chase camera. It's recommended, for the sake of reusability, to implement each one as a separate component class, named VisualComponent, PlayerInput and ChaseCamera. VisualComponent can be used indepen-

dent of the other two for any static object that needs a visual representation such as a house or a tree. None of these need to be followed by a camera or controlled by the player so there will be no need to include the other respective components. However, when used to compose a player object, ChaseCamera and PlayerInput are used and both have a dependency on the existence of an oriented geometry. They use this geometry respectively as a target to look at or to update based on player commands. VisualComponent already defines a geometry attribute (its loaded model) that fits this need. In Figure 4 this relation is expressed in the form of an object diagram. The player game object has a collection of components, all being concrete subclasses of AbstractComponent and having dependencies between them.

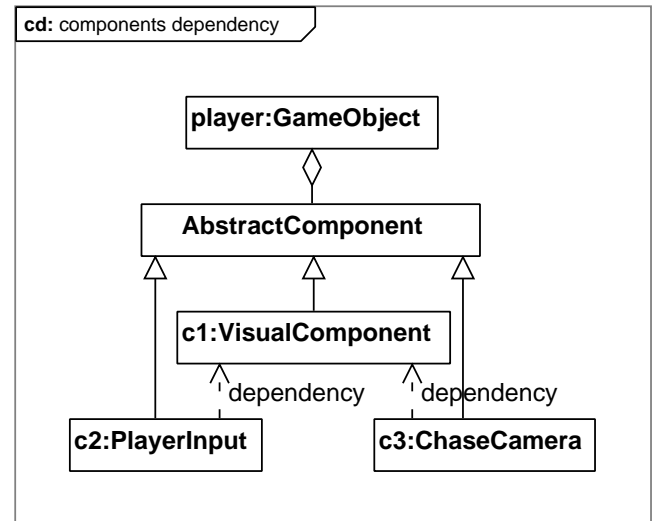


Figure 4: Dependent components example

Components c2 and c3 from Figure 4 show a dependency to component c1. Now it's clear that both ChaseCamera and PlayerComponent have a attribute of type VisualComponent, so they can use it at their respective update methods. Instead of manually looking for this object at each update method, the component programmer only has to include the runtime-available custom annotation @Inject to the attribute declaration as show in Code 5. When initializing each component, as will be explained next, if the annotation @Inject is found before any declared attribute, our framework looks for an instance of this component type in the same game object and sets it into the attribute.

```
class ChaseCamera extends AbstractComponent {
    @Inject
    VisualComponent vc;

    public void update(float interpolation){
        camera.lookAt(vc.getWorldTranslation());
    }
}
```

Code 5: @Inject in ChaseCamera source code

One can see that there is no line of code looking for the VisualComponent instance or checking its nullability. It's also easy to notice that, compared to the example in Code 1 in Page 1, the above code is smaller, considerably cleaner and also safer since the framework will stop initialization and show an error log if there is no VisualComponent declared and already initialized for the related GameObject. In Code 6 one can see a correct XML for the player object composition.

Since this declaration includes a VisualComponent, the other two components, which depend on the existence of the previous, are initialized correctly. Game object initializations occurs as shown in the sequence diagram presented in Figure 5. As shown in the

```

<object name="player">
  <component class="VisualComponent">
    <model value="knight.md5" />
  </component>
  <component class="PlayerInput" />
  <component class="ChaseCamera" />
</object>

```

Code 6: Correct player object composition

diagram, when creating a game state, all game objects and their components are first instantiated and their declared attributes set. After this first step (messages 2, 3 and 4 in the diagram), the game objects are properly initialized and components dependencies are injected (messages 5 and 6). By running the injection last, all declared components are already attached to the game object leading to a complete dependency solving.

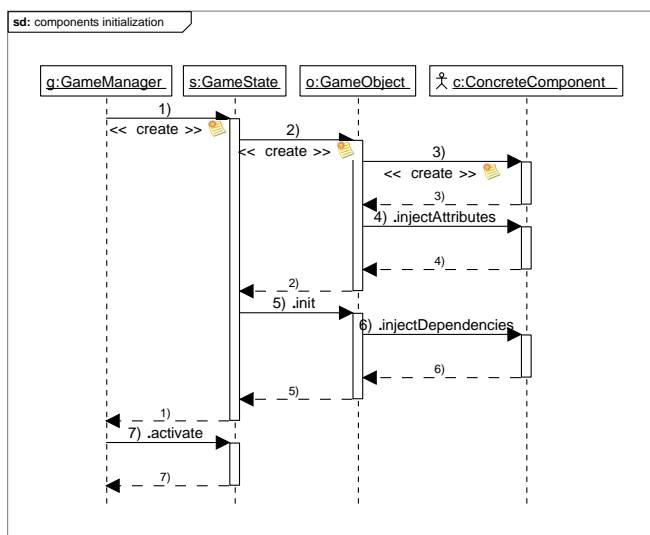


Figure 5: Game objects initialization

In Code 7 an incorrect player object composition is presented. Let's imagine for instance that the level designer made a mistake and thought the "character" supertype already had a VisualComponent declared and included only the (in his mind) two missing ones. Instead of leading to an unpredicted runtime error, this specification is not valid and our framework will show a message at initialization such as seen in Code 8.

```

<object name="player" type="character">
  <component class="PlayerInput" />
  <component class="ChaseCamera" />
</object>

```

Code 7: Incorrect player object composition

```

"Incomplete composition of object: 'player'.
Missing required 'VisualComponent'
needed by included 'ChaseCamera'."

```

Code 8: Unsolved dependency initialization error

By having automatic handling of the coupling between components and a safer initialization of game objects, the programmer will not need to manually check for explicit dependencies or their nullability. From this example one can see that the level design step in

the production pipeline is improved with less dependency on code debugging tools.

4.2 Example 2: Fast Prototyping of New Game Mechanics

During the paper introduction we pointed that one of the problems caused by the dependency between game objects and components was the high number of lines of code dealing with issues not related to the core game logic being implemented. Boilerplate code like this needs a lot of concentration from the programmer and, usually, also debugging. In this example we show how Dependency Injection improves fast prototyping of components by making the programmer focus only on the core mechanics implementation.

Lets assume we are going to implement a "lunar cargo" game, where the core mechanics consists of controlling a rocket-powered heavy weight lift vehicle. The goal of the implementation is to expose the core mechanics to early tests as fast as possible. The following features are essential to the prototype:

1. A moon-like terrain;
2. Gravity and collision physics;
3. 3D model loading;
4. Player-controlled thrust to the lunar module (most important).

It is easy to notice that features 1-3 are also needed by several different game genres and are already available as reusable GCore components. The only lasting, and important, feature is the player-controlled thrust. In Code 9 one can see the complete XML used for this prototype, declaring a single game state, composed of two game objects: the terrain and the lunar module. The implementation of the missing Thrust class is explained next.

```

<game name="lunarCargo" init="moon">
  <gamestate name="moon">
    <!-- prototype object1: terrain -->
    <object name="terrain">
      <component class="TerrainComponent">
        <heightmap value="moon.png" />
      </component>
      <component class="TerrainPhysics" />
    </object>
    <!-- prototype object2: lunar module -->
    <object name="module">
      <component class="VisualComponent">
        <model value="cargo-ship.3ds" />
      </component>
      <component class="DynamicPhysics" />
      <component class="Thrust" />
    </object>
  </gamestate>
</game>

```

Code 9: Lunar Cargo prototype XML

GCore's DynamicPhysics component has a dependency to VisualComponent that the former uses as collision geometry and also to move as the simulation goes. The Thrust being implemented clearly has a dependency to the DynamicPhysics, which is going to have forces applied to as the user presses the "thrust" key. In Code 10 one can see the implementation of the Thrust class. The dependency to DynamicPhysics is exposed to the framework by the attribute annotated with @Inject and the update method just takes for granted that this attribute will not be null at runtime.

It is clear that the programmer could concentrate on the core mechanics implementation: checking for the user input and imposing a thrust to the lunar module physics. With this approach we believe

```

class Thrust extends AbstractComponent {

    @Inject
    DynamicPhysics physics;

    public CargoController() {
        KeyManager.set("thrust", KEY_SPACE);
    }

    public void update(float tpf) {
        if (KeyManager.isValidCommand("thrust"))
            physics.addForce(Vector3f.UNIT_Y);
    }
}

```

Code 10: Thrust component source code

one can write better code and achieve faster prototyping in a game production pipeline without compromising a good design.

5 Conclusion

Data driven is a proved approach to manage risk during a game production pipeline. Putting this together with a well-designed game engine, a powerful architecture is achieved, as can be shown by several successful engines and frameworks. However, object orientation, especially components composition, has some issues with maintainability of highly dependent instances. In this paper we have presented a technique based on the Dependency Injection design pattern that safely removes this task from the programmer duties.

There is a know myth in game production circles that it is impossible to fast prototype and write well-designed code at the same time. We firmly believe that by using the GCore framework one can achieve very fast prototyping without giving up most good programming practices. By using Dependency Injection we completely removed the hard-coded implicit dependencies, which are so common in game objects scripting, from GCore components library, providing a powerful, extensible and safer tool.

GCore is a constant work in progress and we are now investigating how to apply the technique to other scenarios in game design such as dependencies between a component and any attribute from other game objects and components. We plan to extend the use of annotation to enable the programmer to apply constraints, such as not-null, min/max values/length, to any primitive type attribute (plus files, strings, vectors and quaternions). The short-term roadmap also includes a level-editor, which provides for the visual composition of games freeing the level designer from the need to write XML specifications.

Acknowledgements

We would like to thank Scott Bilas for the meaningful discussions about game object components dependency and other relevant issues related to software engineering and current trends in game engines development.

We are also very thankful to everybody over the JMonkeyEngine discussion forums, specially the developers, always willing to help with the accurate solutions on rendering, audio and physics that made the GCore framework possible.

References

3DVIA. Virtools. <http://www.virttools.com/>.

BILLAS, S., 2002. A data-driven game object system. Talk at the Game Developers Conference '02.

BILLAS, S. 2003. The continuous world of dungeon siege. In *Proceedings of the Game Developers Conference '03*.

BILLAS, S., 2007. Optimizing the development pipeline - tools, technology, process. Lecture at the CGA Casual Connect Europe: West 2007.

CRYTEK, 2008. Cryengine sandbox 2 manual. <http://doc.crymod.com/SandboxManual/>.

DE MORAES ZAMITH, M. P., CLUA, E. W., CONCI, A., MONTENEGRO, A., PAGLIOSA, P. A., AND VALENTE, L., 2007. Parallel processing between gpu and cpu: Concepts in a game architecture.

EPICGAMES, 1998. Unrealscript language reference. <http://unreal.epicgames.com/UnrealScript.htm>.

FOLMER, E. 2007. Component based game development - a solution to escalating costs and expanding deadlines? In *Component Based Software Engineering*, Springer, vol. 4608 of *Lecture Notes in Computer Science*, 66–73.

FOWLER, M., 2004. Inversion of control containers and the dependency injection pattern. <http://www.martinfowler.com/articles/injection.html>.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional.

GARAGEGAMES. Torque game engine. <http://www.garagegames.com/>.

HALLER, M., ZAUNER, J., AND HARTMAN, W. 2002. A generic framework for game development. In *In Proceedings of the ACM SIGGRAPH and Eurographics Campfire '02*, ACM.

IMAGINATION, C. Jgn: Java game networking. <http://forum.captiveimagination.com/index.php/board,4.0.html>.

JIN, K., 2007. Why and what of inversion of control. <http://www.pocomatic.com/docs/whitepapers/ioc/>.

JMEPHYSICS. Jmephsics: interface between jme and physics engines. <https://jmephsics.dev.java.net/>.

JMONKEYENGINE. Jmonkey engine 1.0 online documentation. <http://www.jmonkeyengine.com/>.

MICROSOFT. Microsoft xna. <http://www.xna.com/>.

POCOMATIC, 2007. Pococapsule/c++ ioc and dsm framework. <http://www.pocomatic.com/docs/whitepapers/pococapsule-cpp/>.

PONDER, M. 2004. *Component-Based Methodology and Development Framework for Virtual and Augmented Reality Systems*. PhD thesis, Ecole Polytechnique Federeale de Lausanne.

SPINOR. Shark 3d real time 3d software. <http://www.shark3d.com/>.

STOY, C. 2006. Game object component system. In *Game Programming Gems 6*, Charles River Media, M. Dickheiser, Ed., 393–403.

SWEENEY, T. 2006. The next mainstream programming language: a game developer's perspective. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, New York, NY, USA, 269–269.

UNITYTECHNOLOGIES, 2008. Unity3d game engine 2.1. <http://unity3d.com/unity>.