Software Reengineering
Assignment 1

Joey Ezechiëls (1338994) Volker Lanting (1513273)

December 3, 2012

Contents

1 Initial Questions

In reverse-engineering the JMonkeyEngine, there are some initial questions that need answering in order to gain a clearer picture of what the architecture and code look like.

1.1 Main features of the program

jMonkeyEngine can be used to create games. To this end it offers a number of features, among which we consider the following ones the most important:

- Creating Applications (`com.jme3.app`, `com.jme3.system`)
- Managing Assets of these applications (`com.jme3.asset`) The main assets are scenes, located in the `com.jme3.scene` package.
- Real time Rendering (`com.jme3.renderer`)
- Collision detection (`com.jme3.collision`, `com.jme3.bounding`)
- Shaders (`com.jme3.shader`)

1.2 Important source code entities

Feature	Important source entities
Managing assets	AssetManager, ImplHandler, Node, Mesh, Geometry, Spatial
Creating applications	SimpleApplication
Managing/configuring applications	AppSettings, JmeSystem, JmeSystemDelegate, AbstractAppState, AppStateManager
Real time rendering	RenderManager, Camera, ViewPort, RenderQueue
Shaders	Shader, Uniform, UniformBindingManager
Collision Detection	CollisionResults, BoundingVolume

The actual Renderer is located in the lwjgl subsystem and not part of the core subsystem. Therefore we will skip them for this project.

1.3 Quality, Design and Implementation: First impressions

Our first impressions are that generally the package structure is well-defined and the code itself uses a lot of interfaces, which generally is a good sign (though of course not a conclusive indicator of good design).

However there are also some odd design decisions (e.g. an inner enum in a class that is referenced from a class in another package) design violations (perhaps due to organic growth of the code base), and the code base also shows signs that a tradeoff between execution speed and design has been made, particularly in the `com.jme3.math` package.

Generally the documentation seems fairly good and up to date (e.g. there is a slide show on the website explaining how Scene Graphs work in JME3). However in some parts of the code itself there seems to be lack of comments, which would be helpful especially in the more obscure parts of the code base.

Counting in absolute numbers, there is a fairly large set of tests. However it interesting and somewhat disappointing that for the entities we consider major there are almost no tests. For example, the `com.jme3test.math` and `com.jme3test.scene` packages contain only **1** and **2** tests, respectively. Next to that it is quite surprising that the test packages are different from the src packages (i.e. `com.jme3test.math` is used for tests rather than `com.jme3.math`). While it is possible that there are some considerations we don't know about, we consider it a rather dumb decision as it makes testing package-private classes quite a bit more difficult.

1.4 Reengineering feasibility

Reengineering is feasible, as the current design and quality really isn't bad. Also, since we don't have a specific goal for our reengineering (like making it easier to add a specific feature), we can freely choose what parts to reengineer. This makes it obvious that reengineering is possible.

1.5 Exceptional entities

In this section we list some exceptional source entities based on several metrics. The metrics are printed in italics.

Complexity per class	
Package	**Classes**
`com.jme3.math`	FastMath, Quaternation, Matrices, Vectors
`com.jme3.bounding`	BoundingBox, BoundingSphere
`com.jme3.scene`	Node, Mesh, Geometry, Spatial
`com.jme3.renderer`	RenderManager, Caps
`com.jme3.material`	Material, MatParam
`com.jme3.collision`	SweepSphere
Lack of cohesion (LCOM4)	
Package	**Classes**
`com.jme3.system`	AppSettings, JmeSystemDelegate
`com.jme3.math`	Quaternation, ColorRGBA
`com.jme3.scene`	Node, SimpleBatchNode, Spatial
`com.jme3.shader`	Shader
`com.jme3.asset`	AssetKey, ImplHandler
Too much responsibility (Afferent Coupling)	
Package	**Classes**
`com.jme3.math`	Vector3f, FastMath, Quaternation, ColorRGBA, Matrix4f, Matrix3f
`com.jme3.scene`	Spatial, Mesh, VertexBuffer, Geometry, Node
`com.jme3.renderer`	Camera, Viewport Renderer, RenderManager
`com.jme3.asset`	AssetKey, AssetManager
Instability (Efferent/Afferent coupling ratio)	
Package	**Classes**
`com.jme3.asset`	DesktopAssetManager
`com.jme3.scene`	BatchNode
`com.jme3.app`	SimpleApplication
`com.jme3.collision`	BIHTree, BIHNode
`com.jme3.shader`	Uniform, UniformBindingManager
`com.jme3.bounding`	BoundingBox, BoundingSphere

The `com.jme3.math` package has quite some duplicate code blocks. Exceptional classes are LineSegment and Matrix4f.

Complex methods (CC)		
Package	**Class**	**Method**
`com.jme3.math`	LineSegment	distanceSquared
	Matrix4f	get
		set
		equals
		equalIdentity
`com.jme3.scene`	BatchNode	mergeGeometries
	Mesh	setInterleaved
`com.jme3.scene.shape`	Cylinder	updateGeometry
`com.jme3.shader`	Uniform	setValue
	UniformBindingManager	updateUniformBindings
`com.jme3.asset`	BlenderKey	equals
`com.jme3.material`	RenderState	contentHashCode
	Material	read
	MatParam	getValueAsString

Also, a number of **God classes**, **Data classes** and a number of **Feature envy** and **Code duplication** occurrences have been found The 5 most egregious of each of these are:

Data classes	TempVars (50.5), RenderContext (19.25), Environment (14.25), TouchEvent (8.55), StringBlock (8.33)
God classes	Material (81.03), Camera (35.37), RenderManager (33.99), BatchNode (24.96), BufferUtils (23.62)
Feature Envy	BoundingBox.intersects (22), BIHTree.createBox (20) Ray.intersects (19), Camera.lookAt (15), TangentBinormalGenerator.linkVertices (14)
Code Duplication	LineSegment.distanceSquared (9.1), Matrix4f.get (7.4) Matrix3f.get (7.4), Texture3D.setWrap (6.9), TextureCubeMap.setWrap (6.9)

1.6 Inheritance structure

Below are the most important parent classes that we found, along with a short description on where they are implemented or inherited.

Package	**Entity**	**Subtyped in**
`com.jme3.app.state`	AbstractAppState	states in `com.jme3.app`
`com.jme3.asset`	AssetKey	keys in `asset`, `shader`, `scene`, `audio`
	AssetProcessor	processors in `asset`, `material`, `texture`
	CloneableSmartAsset	Material in `material`, Spatial in `scene` Texture in `texture`
	AssetLoader	loaders in `asset`, `material`, `texture`, `scene`, `audio`, `font`, `cursoris`, `shader`, `export`
	AssetLocator	locators in plugins
`com.jme3.collision`	Collidable	AbstractTriangle, Ray in `math`, Spatial in `scene`, BoundingVolume in `bounding`, SweepSphere in `collision`
`com.jme3.scene`	Spatial	inherited by Node and Geometry from `scene`, which are in turn inherited throughout the system
	Mesh	used in all kinds of shapes in `scene.shape`, `scene.debug` and `effect`
`com.jme3.scene.control`	Control	implemented by AbstractControl in `scene.control`, which is in turn extended in all kinds of Controls throughout the system

1.7 Scene composition: Basic elements

Scenes are represented as a scene graph. This graph exists of Spatials, which represent (collections of) objects in space.

There are two types of Spatials: Node and Geometry. Nodes group other Spatials (their children) together, so all these children can be placed and moved relative to their parent. A Geometry node represents an actual visible object in the scene. It can not have children, but does have a Mesh (its shape in space) and a Material (determining its appearance).

1.8 Scene rendering

The RenderManager is responsible for managing the rendering. It has a renderGeometry method, which calls the render method of the Material of the Geometry. This render method will render the Geometry, based on the Material's (render- and shader)state and the Geometry's Mesh (shape). The actual rendering is done by the Renderer that the Material obtained from the RendererManager.

1.9 Collision detection

Geometries have a Mesh, which determines their shape. A Mesh has a BoundingVolume, which determines the boundaries of the Mesh and is used in collision detection. A BoundingVolume must implement Collidable, which means they have a collideWith method. This method is used to check if the current Collidable is in collision with a given other Collidable. It generates a CollisionResult for each collision and adds them to a given CollisionResults object. Then it returns how many collisions were found.

The abstraction does not work as pretty as intended, since the collideWith method is implemented for each different Collidable.

2 Problem detection

By utilising a number of tools available to us (see section 3 for details) and knowledge gained from this course we have identified a number of shortcomings in both the design and implementation of JMonkeyEngine. In searching for these shortcomings we have kept in mind the S.O.L.I.D. design principles, though we have also considered the ADP and DRY principles.

2.1 Single Responsibility violation

2.1.1 JmeSystem

JmeSystem seems to have too much responsibilities. The lack of comments makes it hard to say anything about what the intended responsibility is, but it seems to be: Manage permissions related to IO (isLowPermissions, getStorageFolder), do image IO (writeImageFile), manage system dependent data (getPlatform), manage screen dialogs (setSoftTextDialogInput, showErrorDialog) and serve as a factory for system related managing classes (newAssetManager, newAudioRenderer, createImageRaster).

There is clearly too much responsibility here, which is indicated by Sonar's LCOM4 metric. This metric groups methods based on common attributes. If there are two or more disjoint groups, there is a lack of cohesion between the methods of a class. Usually, this is an indicator that there are two or more responsibilities for the class.

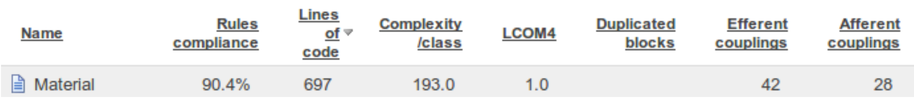
The LCOM4 method is easy to fool, as abstract methods and loggers shared in methods can throw off the LCOM4 measure. The JmeSystem takes all its functionality from the abstract JmeSystemDelegate, which has an LCOM4 of 2, and its subtype JmeDesktopSystem also has an LCOM4 of 2.

Inspection of some of the afferent couplings of JmeSystem leads to the following usage table, illustrating how breaking up the class will result in less coupling.

Part	Usage
factory	AwtPanelsContext
	Application
	ShaderCheck
	ImageRaster
dialog	InputSystemJme
	Application
	SimpleApplication
permissions	ClasspathLocator
	AppletHarness
	LwjglAbstractDisplay
image IO	ScreenshotAppState
storage folder	SaveGame
	ScreenshotAppState
platform	LwjglCanvas

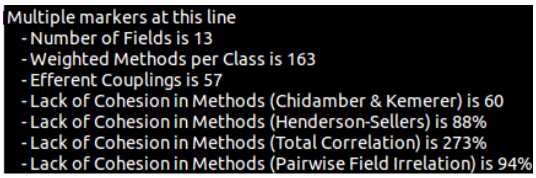
2.1.2 Material

Material is quite an exceptional entity, with an efferent coupling of 42, 697 lines of code, using 77 accessors or attributes of 24 unrelated classes, having a complexity of 193 according to Sonar's check, and triggering all four lack of cohesion of methods indicators of Eclipse Metrics.



Name	Rules compliance	Lines of code	Complexity /class	LCOM4	Duplicated blocks	Efferent couplings	Afferent couplings
Material	90.4%	697	193.0	1.0		42	28

Figure 1: some Sonar metrics for Material



Multiple markers at this line
- Number of Fields is 13
- Weighted Methods per Class is 163
- Efferent Couplings is 57
- Lack of Cohesion in Methods (Chidamber & Kemerer) is 60
- Lack of Cohesion in Methods (Henderson-Sellers) is 88%
- Lack of Cohesion in Methods (Total Correlation) is 273%
- Lack of Cohesion in Methods (Pairwise Field Irrelation) is 94%

Figure 2: some Eclipse Metrics results for Material

It has gotten so big, complex and coupled that it is hard to figure out what it is doing, and even harder to believe it is a single responsibility. According to the javadoc comments, it has the following responsibility: Material describes the rendering style for a given Geometry. However, a Material is not specific for a given Geometry. Nor is it purely a collection of parameters to configure the style of rendering. We think the real responsibilities of Material are: describe a rendering style (MatParam, RenderState, Technique, transparent, receiveShadows), and apply it to a Geometry and render it (updateLightListUniforms, preload, render).

2.2 Open-Closed Principle violation

Using Eclipse metrics we encountered the `com.jme3.material.MatParam` class and in particular its method `MatParam.getValueAsString()`, which has a cyclomatic complexity of **19**, consists of **124** SLOC and yet has only **60** statements. These odd numbers led us to investigate the class.

We found that the enum `com.jme3.material.MatParam` and the class `com.jme3.shader.VarType` violate the Open/Closed Principle. `MatParam.getValueAsString()` uses a switch-case to check for certain "types", which are encoded as a field within `MatParam`. The type of the field is `VarType`. The problem with this aproach is that both the list of types in the switch-case and the list of enums within `com.jme3.shader.VarType` is explicit and finite, so when a new "type" is created the existing code has to be modified not in 1 but in at least 2 places at once in order to stay in sync.

2.3 Dependency Inversion Principle violation

The layer for rendering should be a higher layer than the layer with materials. Therefore, the dependency inversion principle requires the material layer to implement an interface of some sort in the renderer layer. However, in the current system the `com.jme3.renderer` package depends on the `com.jme3.material` package. Or more specific: the RenderManager uses the Material class directly. As can be seen in Figure 3.

This means that changes to the Material class will propagate to the RenderManager and it is not easy to replace the current implementation of Material.

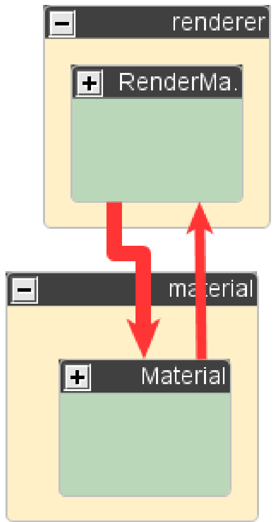


Figure 3: A DIP violation

2.4 Acyclic Dependency Principle violation

The packages `com.jme3.material`, `com.jme3.renderer` and `com.jme3.scene` together violate the ADP on the package level:

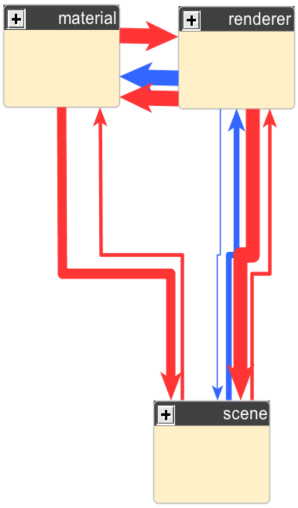


Figure 4: An acyclic dependency principle violation. Red arrows represent method invocations while their thickness represents the number of invocations

They call each other's methods: In one direction (the 3 packages actually form 2 cycles), **26** methods contained in classes in `com.jme3.material` are invoking methods in classes in `com.jme3.renderer`, **33** methods in classes in `com.jme3.renderer` are calling methods contained in classes in `com.jme3.scene` and **4** methods contained in classes in `com.jme3.scene` are calling methods contained in classes in `com.jme3.material`.

The reason the ADP violation is a problem is that the packages become tightly coupled, which means that deployment becomes more difficult as now at least these 3 (and possibly more) packages cannot be independently deployed. Modifying the classes within these packages also becomes more difficult as there is no coherent interface to the subsystems the packages represent while at the same time the consequences of these modifications become harder to predict.

2.5 Don't Repeat Yourself violation

The function `distanceSquared(LineSegment)` in the package `com.jme3.math.LineSegment` clearly violates the DRY principle, as the same code structure is repeated over and over again. As it turns out, the LineSegment class contains **24** of such duplicate code blocks while the cyclomatic complexity of the method is **37**. Furthermore, the method is **284** SLOC long and has **197** statements in it. Together these extremely high numbers indicate that the method is difficult to understand and therefore difficult to maintain.

The following snippet was taken from Region 1 according to the accompanying comment, and runs through lines 218-222. The snippet looks like this:

```
tempS1 = -(negativeDirectionDot * s0 + diffTestDot);
if (tempS1 < -test.getExtent()) {
  s1 = -test.getExtent();
  squareDistance = s1 * (s1 - (2.0f) * tempS1)
  + s0 * (s0 + (2.0f) * diffThisDot)
  + lengthOfDiff;
```

The `if`-statement is not closed because line 223 starts an `else if`-statement, which is a duplicate form of this one w.r.t. structure.

3 Used tools

Below is a list of the analysis and refactoring tools we used.

i) **inCode** is an Eclipse plugin that generates *class blueprints* from Java source code.

ii) **Sonar** is a tool which entails several code checking tools like PMD and FindBugs. It also checks for metrics like complexity, code duplicates and coupling. The findings are easily browsable via the Sonar server.

iii) **Eclipse Metrics** is and eclipse plugin that checks for a collection of metrics. It places markers at the source code, but can also export its findings to a browsable HTML format.