The Guerrilla Guide to Game Code

Jorrit Rouwé – Lead Programmer Shellshock Nam '67 – Guerrilla Games
Published: 14 April 2005 on Gamasutra

Introduction

There are a lot of articles about games. Most of these are about particular aspects of a game like rendering or physics. All engines, however, have a binding structure that ties all aspects of the game together. Usually there is a base class (Object, Actor or Entity are common names) that all objects in the game derive from, but very little is written on the subject. Only very recently a couple of talks on gameltech have briefly touched on the subject [Bloom], [Butcher], [Stelly]. Still, choosing a structure to build your game on is very important. The end user might not "see" the difference between a good and a bad structure, but this choice will affect many aspects of the development process. A good structure will reduce risk and increase the efficiency of the team.

When we started Shellshock Nam '67 (SSN67) at Guerrilla Games we were looking for a structure that would be flexible enough to handle all of our ideas, yet strict enough to force us into a structured way of working. The development of our first game Killzone was already well underway so we had a good opportunity to look at their structure and improve on it. After a couple of weeks the base structure for our game had been designed and over the course of the next 2 years a lot changed but our basic structure remained more or less the same.

The core design decisions we made at the start of SSN67 are:

- The system needs to be (mostly) data driven
- It should use the well known Model-View-Controller pattern
- The game simulation should run at a fixed update frequency to ensure consistent behavior on all platforms (PS2, XBOX and PC)

In the remainder of this article these points will be worked out to show how they were implemented in SSN67.

Entities

In SSN67 the base class for all game objects is called Entity. We make a clear distinction between objects that have game state and those that don't. For example, the static world geometry and its collision model are not Entities, neither is a moving cloud in the sky. The player can't change the state of these objects so they are not an Entity. An oil barrel, for example, is an Entity because when it explodes it can harm the player and therefore influences the game state.

Using a definition like this makes it very easy to separate objects that are important to the game from objects that are not. Our streaming system, for example, can stream textures

and other rendering data in and out without affecting the game state. Another system that benefits is the save game system. The state of an Entity is saved and on reload all current Entities are destroyed and replaced by Entities from the save game. Because all static geometry is unaffected saving and loading is very quick and the resulting save game is small. To go back to our example of a moving cloud: In SSN67 clouds are not Entities and therefore not saved. If you look carefully you can see this.

Most of the Entities in SSN67 need a position and orientation in the world so our Entity by default contains a 4x4 matrix. For those Entities that do not need a position we just accept the overhead.

Data driven

Data driven systems use data instead of code to determine the properties of a system where possible. In SSN67 we used the EntityResource class as base class for Entity properties. EntityResources are defined in a text file and edited by hand. The following example shows what the configuration for an NPC could look like:

```
CoreObject Class=HumanoidResource
{
        Name = Ramirez
        Model = models/soldiers/ramirez
        Speed = 5
        …
}
```

When these text files are loaded by our serialization system the corresponding objects are automatically created and each variable is mapped onto a member variable. After an EntityResource is loaded we validate the range of each variable and check if all variables together form a correct configuration. If this is not the case the game will display an error and refuse to run until the problem is corrected.

In SSN67 we do not allow Entities to be created without an EntityResource. Using an EntityResource makes sure that a specific object acts the same in all levels. This generally reduces the amount of bug fixing when the code for an Entity changes, because there is no need to test every level but only the EntityResources that are affected. We use a simple folder structure to store our EntityResources with one object per file.

The Factory pattern [DP] is used to create an Entity from its EntityResource. This means that if you have an EntityResource you can create a corresponding Entity ready to be used in game. We use the factory mechanism, for example, in our Maya plug-in. Level designers can create an Entity in a level by selecting the corresponding EntityResource file. Maya uses the factory to create an Entity for it and allows the level designer to position it. Entities that need specific (dynamic) behavior are controlled from our scripting language LUA [LUA]. The properties that can be set from LUA usually boil down to giving high level commands to the AI so that most of the configuration of an Entity is still contained in the EntityResource. Supporting a new Entity class in Maya is as simple as recompiling the Maya plug-in.

Model-View-Controller

The Model-View-Controller pattern [DP] is used in many areas. In a word processor, for example, the Controller handles mouse and keyboard input and translates these to simple actions. The Model manages the text and executes the actions dictated by the Controller. The View is responsible for drawing (a portion of) the text on screen and communicates with the display drivers.

In a game we can use the same pattern. In our case the Model is called Entity, the View is called EntityRepresentation and the Controller is called Controller. We will now discuss the roles of each of these classes in game.

The Model (Entity)

Entities try to keep a minimal state of the object. Any state that is only needed for visualization is not part of the Entity. Most Entities in SSN67 are simple state machines.

Ideally the Entity should not know about its animations but unfortunately most animations affect the collision volume of the Entity and therefore the game state. Still, we try to keep Entities and their animations as loosely coupled as possible. A walking humanoid, for example, looks at its EntityResource for its maximum walk speed and not at the walk animation. The EntityResource precalculates the speed of the walk animation and speeds up / slows down the animation based on the current speed. This makes it possible to use a simple model for a walking humanoid (linear movement) while the animation always matches the current speed perfectly.

The View (EntityRepresentation)

The EntityRepresentation is responsible for drawing the Entity and all of its effects.

Examples of things that an EntityRepresentation manages are:

- 3D models
- Particles
- Decals
- Screen filters
- Sound
- Controller rumble
- Camera shake

An EntityRepresentation can look at but not change the state of the Entity. The current state of an Entity is usually enough to calculate the state of the EntityRepresentation. In some cases this is not practical however. For example, when a human gets hit we want a blood spray from the wound. Simply looking at the previous health and the current health does not tell us where the enemy was hit. For these cases we use a messaging system. All messages derive from a class called EntityEvent, which contains an ID that indicates the

type of the message. The getting hit message also contains the type and amount of damage and where the hit occurred. The Entity fills in the structure and sends it to a message handler in the EntityRepresentation. It won't receive a response to this message so the communication is one way only.

To achieve a game object that looks realistic the timing of effects is usually critical and needs a lot of code to manage. By separating this code from the essential game logic we reduce the risk of bugs. Another advantage is that if all objects follow the rules, the game state is not dependent on any of the EntityRepresentations and the game can run without them. This principle can be very useful for creating a dedicated multiplayer server where graphical output is not needed.

The Controller

The Controller provides abstract input for an Entity. For every Entity that supports Controllers we create a base class that defines the controllable parts of the Entity. From this version we can derive an AI and a player version of the Controller. For example, a Humanoid has a HumanoidController and we derive a HumanoidAIController and a HumanoidPlayerController from this controller. The player version of the controller can be further split up to provide mouse and keyboard support (for PC) or joystick support (for XBOX and PS2).

The role of the Controller is to specify what the Entity should do. The Controller can't directly change the state of the Entity. Every update the Controller is polled by the Entity and the Entity tries to follow the Controllers instructions.

For example, the Controller would never call a MoveTo() function on a Humanoid but instead the Humanoid would poll the Controller's GetDesiredSpeed() function and then try to reach this speed while performing collision detection to make sure not to clip through walls.

The Controller's functions are more abstract than IsButtonPressed() or GetJoystickAxisX(), this to make it easier for the AI to use the Controller.

To make a Humanoid interact with a mounted gun, for example, the HumanoidController implements a function called GetUseObject(). This function is polled every update by the Humanoid. The AI uses reasoning to determine if it is beneficial to use a specific mounted gun. When it chooses to use a mounted gun it walks to the correct location and returns the mounted gun through GetUseObject(). When the player stands next to a mounted gun the HumanoidPlayerController detects that the mounted gun is in range and handles the logic of the use menu. The HUD takes care of drawing the menu. When the selection is made it is returned through GetUseObject(). Next time the Humanoid polls the HumanoidController it will start up the mount process. The HumanoidController is disabled and the mounted gun receives a MountedGunAIController or MountedGunPlayerController so that it can be used.

A Simple Example

We will illustrate the Model-View-Controller mechanism with an example of a tank.

The tank is split up in an Entity (Tank), EntityRepresentation (TankRepresentation) and a Controller (TankController).

The TankController has functions like GetAcceleration(), GetSteerDirection(), GetAimTarget() and Fire(). The AI steers the tank using a TankAIController, the player uses a TankPlayerController.

The Tank keeps track of the physics state of the vehicle. It looks at the controller and applies forces to the physical model and responds to collisions. It keeps track of damage state, turret direction and implements firing logic.

The TankRepresentation draws the tank. It creates sparks particles and corresponding sound when the tank collides. It plays the engine sound and changes its pitch when the speed of the tank changes. It creates smoke particles when the tank fires and produces track marks when the tank is driving. It can also fine tune the position / rotation of the tracks so that they follow the terrain exactly (without influencing the collision volume of the tank).

An Example of Simplifying the Game State

When a human fires a gun, the bullets exit from the muzzle of the gun. In SSN67 this sometimes led to problems. Every weapon had its own aiming animations that were slightly different. These differences led to inconsistencies with our precalculated cover positions for the AI. The AI would sometimes think that they could fire from a specific location but when they got to that position and tried they would shoot into a rock.

To improve the situation we separated the logic of firing from the visual effect (the tracers). The bullets now exit from a fixed offset (roughly where the shoulder is) based only on the current position and stance of the humanoid. The tracers are handled by the EntityRepresentation and come from the muzzle of the gun (which is taken from the animation) and move in the same direction as the bullet. They slowly converge towards the real path of the bullet so that the impact point and time for the bullet and tracer is the same.

For the player it turned out that with our 3rd person camera it sometimes looked like you could fire over a piece of cover where in reality you couldn't. We tweaked the firing offsets for the player so that he was shooting from eye height instead of shoulder height and fixed the problem.

This simple model in the Entity is easy to use for the AI and guarantees consistency throughout many bits of code that are related to aiming and firing. It trades complicated

code throughout the system for complicated code localized in the EntityRepresentation where it can only influence the EntityRepresentation itself.

The complete framework

In Figure 1 you can see the complete framework used in SSN67.

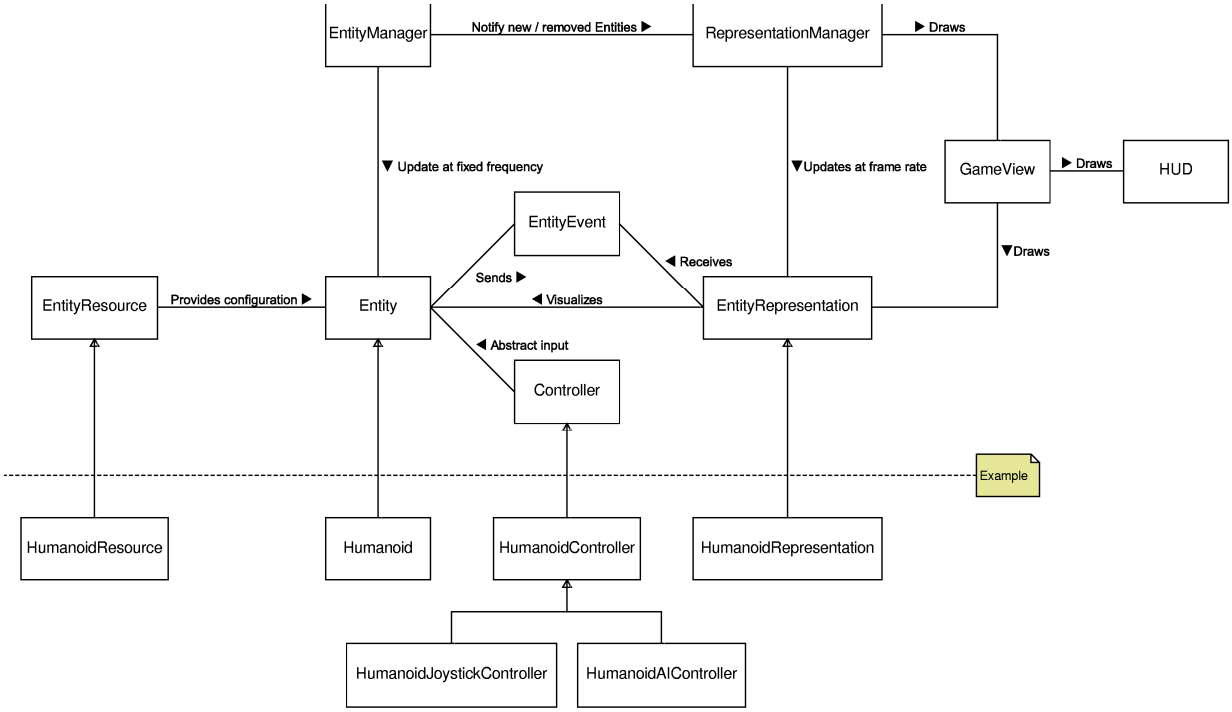

Figure 1: Class diagram of the most important classes and their relations. The classes needed for a Humanoid are provided as an example.

The EntityManager is a container for all Entities. Entities that participate in the game must be added to the EntityManager. The EntityManager updates Entities at a fixed frequency (see next section) and facilitates searching.

We allow Entities to be updated by other Entities in special cases where the update order matters. When a player mounts a mounted gun it is essential that the player is updated after the mounted gun or else the system will jitter. In this case the mounted gun can set a flag on the player so that it will not receive updates from the EntityManager and it can update the player itself.

The RepresentationManager keeps all EntityRepresentations in a spatial hierarchy suitable for quick visibility determination. It gets a notification from the EntityManager as soon as an Entity is added or removed and will create or destroy the corresponding EntityRepresentation.

When drawing a frame the RepresentationManager draws all GameViews. A GameView draws the world for one player. It keeps track of the active camera for a player and uses the spatial hierarchy in the RepresentationManager to draw all visible EntityRepresentations. It also draws all non-entities like the static world. After the scene is drawn it draws the HUD as a 2D overlay. SSN67 could be played split screen multiplayer but this feature didn't make it in the final game due to time constraints.

Fixed frequency

The complexity of the scenes in SSN67 didn't allow for a constant 60 frames per second so our target was 30 frames per second on NTSC and 25 on PAL consoles. We chose a constant 15 Hz update frequency for all Entities. Using a fixed frequency update ensures that all algorithms run exactly the same on all platforms.

A 15 Hz update normally requires an update of all Entities every other frame. The EntityManager could have been used to perform load balancing by updating half of the Entities one frame and the other half the other frame. Instead, we opted for a simpler solution: Updated the high level AI system in frames that we do not need to update the Entities. The high level AI system calculates behaviors, does path planning and a lot of ray casting to determine suitable attack / defend positions for the AI.

The balance between the AI and Entities wasn't exactly ideal in SSN67 as the AI usually needed much less time. This led to a stuttering frame rate when playing the game. To solve this problem we did not wait for the VSYNC interrupt in a loop but instead we would use this time to start updating the entities or AI for the next frame.

If the frame rate drops below 15 frames per second (which can occasionally happen) we slow down the game time so that it increases with max $1/15^{th}$ of a second per frame. Not doing this means that you have to do two update cycles in a single frame, which virtually guarantees that this frame is also going to take more than $1/15^{th}$ of a second to calculate. The profiler showed that doing multiple updates in a single frame can pull the frame rate down for a couple of seconds afterwards.

Because we update the Entities at a lower rate than the frame rate there is a need for interpolation to make all movements smooth. At first we were using extrapolation to try to make Entity movement smooth but this led to very bad results. The used approach is to remember the previous Entity state and the current state and blend from the previous state to the current state in $1/15^{th}$ of a second. This means that there is a constant delay of $1/15^{th}$ of a second between what the player does and what the player sees. This time is small and constant so that people don't notice it.

In SSN67 most Entities were using a skinned model. The following pseudo code shows how our system was set up in this case. By choosing a good base or helper class the actual interpolation code only needs to be written once and most Entities do not need to be aware of interpolation.

```
// Function called by the EntityManager to store the previous state
```

```
// of the Entity for interpolation
void ExampleEntity::PreUpdate(float FixedFrequencyTime)
{
   // Store the previous world matrix
   PreviousWorldMatrix = CurrentWorldMatrix

   // Store the previous array of bone matrices
   PreviousBoneMatrices = CurrentBoneMatrices

   // Notify our EntityRepresentation
   EntityRepresentation->PreUpdate(FixedFrequencyTime)

   // Reset changed flag for next frame
   Changed = FALSE
}

// Function called by the EntityManager after PreUpdate to
// update the state of the Entity
void ExampleEntity::Update(float FixedFrequencyTime)
{
   // Update game state (animations, position, etc.)
   …

   // The EnityRepresentation needs to know if anything changed this frame
   // that requires interpolation
   if (CurrentWorldMatrix or CurrentBoneMatrices changed)
      Changed = TRUE
}

// Function called by Entity::PreUpdate to store the previous state
// of the EntityRepresentation for interpolation
void ExampleEntityRepresentation::PreUpdate(float FixedFrequencyTime)
{
   // This flag indicates of for the next 1/15th second the EntityRepresentation needs to interpolate
   MustInterpolate = Entity->Changed

   // Note the current time
   LastChangedTime = FixedFrequencyTime
}

// Function called by the Representation manager to update the state
// of the EntityRepresentation
void ExampleEntityRepresentation::Update(float RealTime)
{
   if (MustInterpolate)
   {
      // Calculate the interpolation fractor, the factor will be in the range [0, 1]
      float Factor = Min((RealTime - LastChangedTime) / FixedFrequencyTimeStep, 1)

      // Use spherical linear interpolation for the world matrix
      WorldMatrix = SLERP(Entity->PreviousWorldMatrix, Entity->CurrentWorldMatrix, Factor)

      // Use linear blending for the array of bone matrices
      BoneMatrices = (1 - Factor) * Entity->PreviousBoneMatrices + Factor * Entity->CurrentBoneMatrices
   }
   else
   {
      // We're not interpolating so we can simply use the current Entity statie
      WorldMatrix = Entity->CurrentWorldMatrix
      BoneMatrices = Entity->CurrentBoneMatrices
   }
}
```

In Figure 2 you can see how these functions are called.

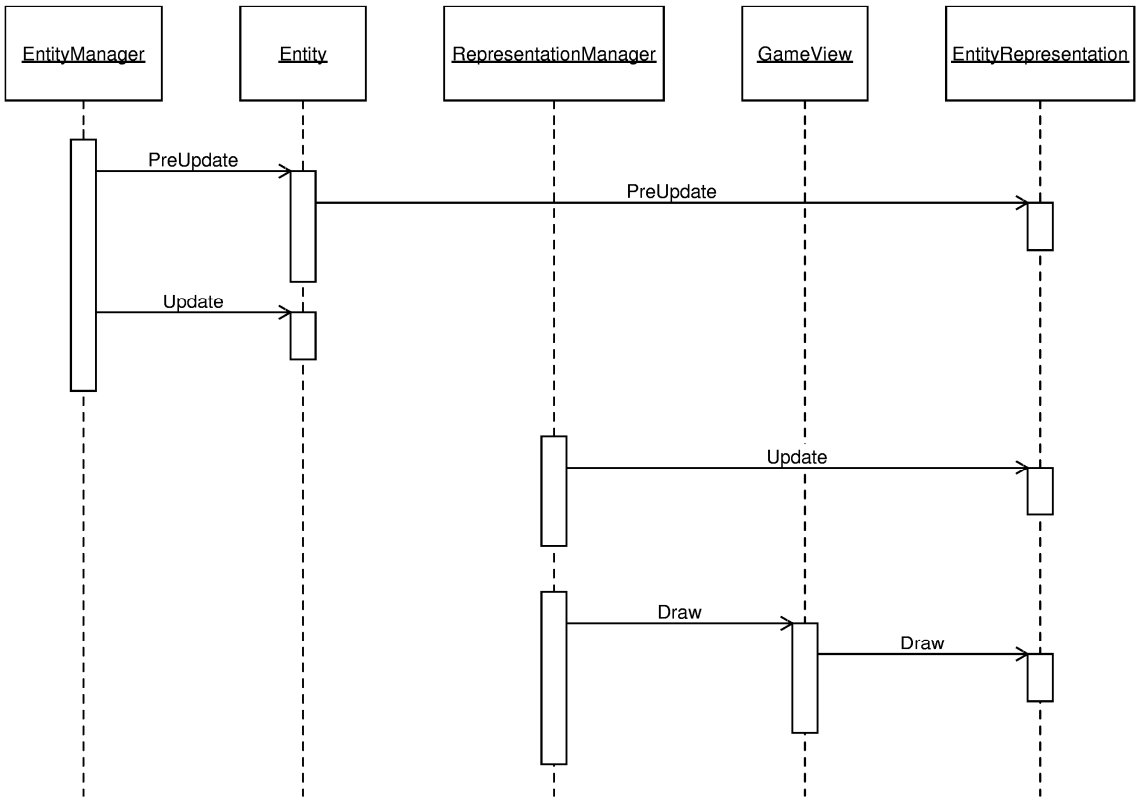

Figure 2: Sequence diagram showing the flow of update and drawing. The PreUpdate function stores the previous state of the Entity / EntityRepresentation for interpolation. The Entity::Update function updates the actual game state. This is all done at 15 Hz. The RepresentationManager updates and draws the representations at full frame rate.

We used spherical linear interpolation (SLERP) for the world matrix and linear interpolation for the bone matrices. Linear interpolation doesn't preserve scale but is a lot cheaper. As long as animations are fairly smooth you won't see the difference.

There are a few simple optimizations to be made to this code to make it practical in real situations. First of all, keeping two sets of world and bone matrices and a reference to the active one saves copying the matrices. Secondly it is possible to do a lazy update so that interpolation is only done when the EntityRepresentation is visible.

Another nice optimization that we used is to update the bounding box for visibility determination at 15 Hz as well. The bounding box is the union of the bounding box for the previous state and the current state. In general this makes the bounding box only a little bit bigger but updating the spatial hierarchy only has to be done at 15 Hz.

We use the interpolation system also for animation LOD. The Entity doesn't update its animations at 15 Hz but at (15 / N) Hz where N is an integer. When the Entity is close to the camera it will use N = 1, when it gets further away N increases. The EntityRepresentation interpolates from its previous animation state to the current animation state in N Entity updates instead of in 1 Entity update. This requires a second

interpolation factor to be calculated in the EntityRepresentation. The rest of the algorithm stays exactly the same so the LOD system practically comes for free. As long as the Entity is loosely coupled to its animations there will be no side effects.

Most common problems in SSN67 with the interpolation scheme were when objects had to be attached to a bone of another object. A gun, for example, would jitter in the hand of a soldier because there was a slight difference between the interpolation of the gun (an Entity) and the hand. We solved this problem by allowing EntityRepresentations to override their world matrix with the interpolated bone matrix from another EntityRepresentation.

Another common problem is when sudden changes in position or animation happen, for example, when a character needs to rotate 180 degrees in one frame. To solve these problems simply turn off interpolation for one update by setting PreviousWorldMatrix = CurrentWorldMatrix and PreviousBoneMatrices = CurrentBoneMatrices.

Conclusion

Using the Model-View-Controller mechanism has a lot of benefits when it comes to separating functionality into manageable blocks. The main drawback is that there will be some code and memory overhead to support an Entity and EntityRepresentation class for every game object.

The interpolation system doesn't come for free, but, looking at our profile sessions interpolation it was a lot cheaper than updating the whole system at the full frame rate. Also, interpolation sometimes leads to visual glitches that require some effort to fix them.

The Model-View-Controller mechanism fits nicely with other often used techniques like making a deterministic game. A deterministic game has a fixed set of input parameters (like controller input + random seed) and when given the same inputs it will reproduce the same results every time. In this case the code that needs to conform to this (Entity) is easily separated from the code that doesn't need to conform (EntityRepresentation). The Controller clearly defines the input parameters of the system.

Another thing that the Model-View-Controller architecture is really good at is online multiplayer. Because Entities maintain minimal state it is much easier to extract the state that has to be sent over the network. Experiments that we did after shipping SSN67 show that this is indeed the case.

It is our opinion that this design played no small part in allowing us to meet every single deadline and milestone on the SSN67 project. For our next project(s) we have started using the same system again.

Acknowledgements

I would like to thank Kasper J. Wessing, Remco Straatman, Frank Compagner, Robert Morcus, Stefan Lauwers and all the other coders at Guerrilla that helped create the SSN67 architecture and this article.

References

- [SSN67]: Shellshock Nam '67 – Developed by Guerrilla, published by Eidos (http://www.shellshockgame.com/)

- [Bloom]: Stranger's Wrath – Charles Bloom – Speech at game|tech 2004 (http://www.cbloom.com/3d/game_tech_04.zip)

- [Butcher]: Halo 2 – Chris Butcher – Speech at game|tech 2004 (http://www.game-tech.com/)

- [Stelly]: Half Life 2 – Jay Stelly – Speech at game|tech 2004 (http://www.game-tech.com/)

- [DP]: Design Patterns: Elements of Reusable Object-Oriented Software - Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (http://www.amazon.com/exec/obidos/ASIN/0201633612/002-1551927-3481617)

- [LUA]: The LUA scripting language (http://www.lua.org/)