Gama Network Presents:

Gamasutra.com

A Software Process for Online Game Tools Development

By Paul Frost

Gamasutra

September 27, 2004

URL: http://www.gamasutra.com/resource_guide/20040920/frost_01.shtml

So, you're going to make a Massively Multiplayer Online Game? You've got a kick ass game design, the hottest graphics engine on the market, the best programmers and designers in the industry, and plenty of budget for development. Sounds great, but what about your software tools? How will the content make its way from the game design specifications into the world? Of course, you've thought about the tools ("We need them!") and you've planned for their development time ("Um, a bit?"), and they're going to be easy to use ("Ah, I think so?") Right?

Asheron's Call 2: Fallen Kings

Developing tools for MMOs is similar to running any other software project, albeit on a smaller scale. In some respects, your tools are mini-client applications that interface to a subset of the game's full functionality. It's important to follow some aspect of standard software methodology when developing tools; throwing them together ad-hoc or as needed is just asking for trouble. Well-defined and developed software tools will stand the test of time. Remember, long after you move on to "the next big thing," your post-launch Live team will be praising (or cursing) your name for months or years to come.

What exactly is "standard software methodology?" There are all kinds of scientific definitions in software development textbooks, but to me it means "a process by which software can be developed in a reliable and repeatable manner." Here at Turbine Entertainment Software, we develop our software tools using five distinct phases: investigation and requirements gathering, design, prototype and implementation, testing, and polishing. Each of these steps sounds fairly self-explanatory; in the next few pages, I will attempt to identify some of the pitfalls and successes Turbine has encountered at each stage of its tools development process.

Investigation and Requirements Gathering

A period of thorough investigation and requirements gathering provides a solid basis for your software project. This step applies when creating new software tools, or extending an existing tool with new features. We use three methods when gathering requirements:

sitting with the users in order to observe their workflow, having the tool developers try their hand using the tools to complete the same work as the user, and brainstorming meetings.

We have found that the most useful method by far is observing users at work. Past sessions illuminated the productivity-halting effect when users were required to switch applications to check files out of revision control. This bottleneck yielded the task: "Integrate automatic revision control checkout functionality into the tools." Trying your hand at a content task while using your own tools is like taking a dose of your own medicine; you will quickly find out what annoying features you may have unwittingly implemented. Brainstorming methods are my least favorite method, unless we've already sat with the users and seen the process. Why? Our Vice President of Engineering, Chris Dyl, summarizes the topic with this pearl of wisdom: "Never tell an engineer what you want, always show them what you are trying to do. Only then will you get a feature that meets your needs." Too often, brainstorming meetings yield lists of unnecessary features, and not lists of problems and end results.

After gathering all of the pertinent requirements, the next phase is to organize them. We do this by determining which features will save the users the most time. Using this "bang for the buck" approach generally associates the most difficult programming tasks with the highest time savings. Removing manual processes and automating lengthy or multi-step processes is the secondary focus: error checking, file copying, auto-generation of code and data, etc. I recently re-read Don Moar's Gamasutra web article "Growing a Dedicated Tools Programming Team: From *Baldur's Gate* to *Knights of the Old Republic*". Don discusses a lot of the issues involved with increasing user productivity. Keeping these guidelines in mind, while generating requirements for our software tools, maintains a focus on productivity savings. Following the organization of requirements, circulate the 'completed requirements' document for signoff. This important step ensures nothing is forgotten.

Admittedly, most engineers would rather not write documentation. Some say it's boring, a lot of typing, and isn't doing anything to help get the "real work" done. Documentation provides the significant functions of organization and communication. Indoctrinating your newly hired engineers into the document writing process not only familiarizes them with the task, it also builds good software practices. When the document generation is built into the process, existing engineers become required to complete it. A second "signoff" step further enforces the behavior. The longer you wait to write documentation in your software process, the harder it will be.

With a solid set of requirements in documented form, scheduling can take place. This bleeds over into the design phase of software development, since we often do some investigation into methods and ideas used by existing code. "Scheduling investigation" gives us a rough idea of how much time any particular requirement or task will take. With tools development, however, it is important to balance feature implementation with bugfixing. Regardless of how well code is written, there will be tweaks and bugs that require immediate fixing. Taking the time to schedule those interruptions will provide a

sane schedule later on. Additionally, build some padding into your schedule. Depending on the perceived difficulty of the tasks, estimated during investigation, anywhere from twenty to fifty percent extra will cover most crises. Plan for this padding. If you find yourself with extra time in the middle of your development schedule, filling the gaps with smaller improvements or "tier 2" features will garner further appreciation from your users.

The Design Phase

The design phase is where your users will find out if you really understood what they were asking for. It is also the last time to change your mind about requirements and functionality in order to avoid a costly rewrite. We like to create two documents at this stage, a high level design signed off by the content users and a detail design for the engineers. The high level design document contains "use cases" and dialog box mock-ups. A "use case" is a description of how the user will use one facet of the functionality the software tool is providing. Our use cases contain a simple description of a single-user action to begin the process and the expected results. Note that the "use case" does not attempt to tell the engineer how the internals of the process should be designed, but it does give the user some idea of what is happening as they perform each action.

Tools development often involves mock-ups of dialog boxes and other user screens. These are quickly prototyped with Microsoft DevStudio's dialog box editor, and included in the design document as screen captures. We often solicit early feedback on these dialog boxes and prototype the look and feel of data entry before adding them to the document. Descriptions of the dialog box fields, error validation, and 'OK/Cancel' button processing is typically included to flesh out the graphics. Like the requirements specification, signoff is received via the high level design document.

The detailed design document is more for the developer than the content users, and as such can be more technically oriented. Many software development efforts skip the detailed design phase, but it is an important one. During the detailed phase, a single "use case" is taken from the high level design document and broken down to the file and module level. This is the time when APIs for new systems are specified, and multiple engineers can brainstorm the best way to implement an optimal solution. Examining existing code for something similar or identifying the need for a common function becomes apparent. By providing early visibility on problems, the detailed design phase allows the engineer to avoid obstacles that would normally constitute roadblocks during the implementation phase. Spending a little time during the design phase to investigate the complexities of the obstacles saves the engineer time and frustration later on.

The debate rages about whether a detailed design document is necessary, or whether you can jump right in and begin coding. Some engineers prefer to iterate over a "prototype-design-prototype-implement" cycle, while others prefer to prototype while creating the design document. Some amount of detailed design is necessary; the detailed design document helps in organizing code flow, indicating where common code is being written, documenting the task at a source file and module level, and providing a roadmap

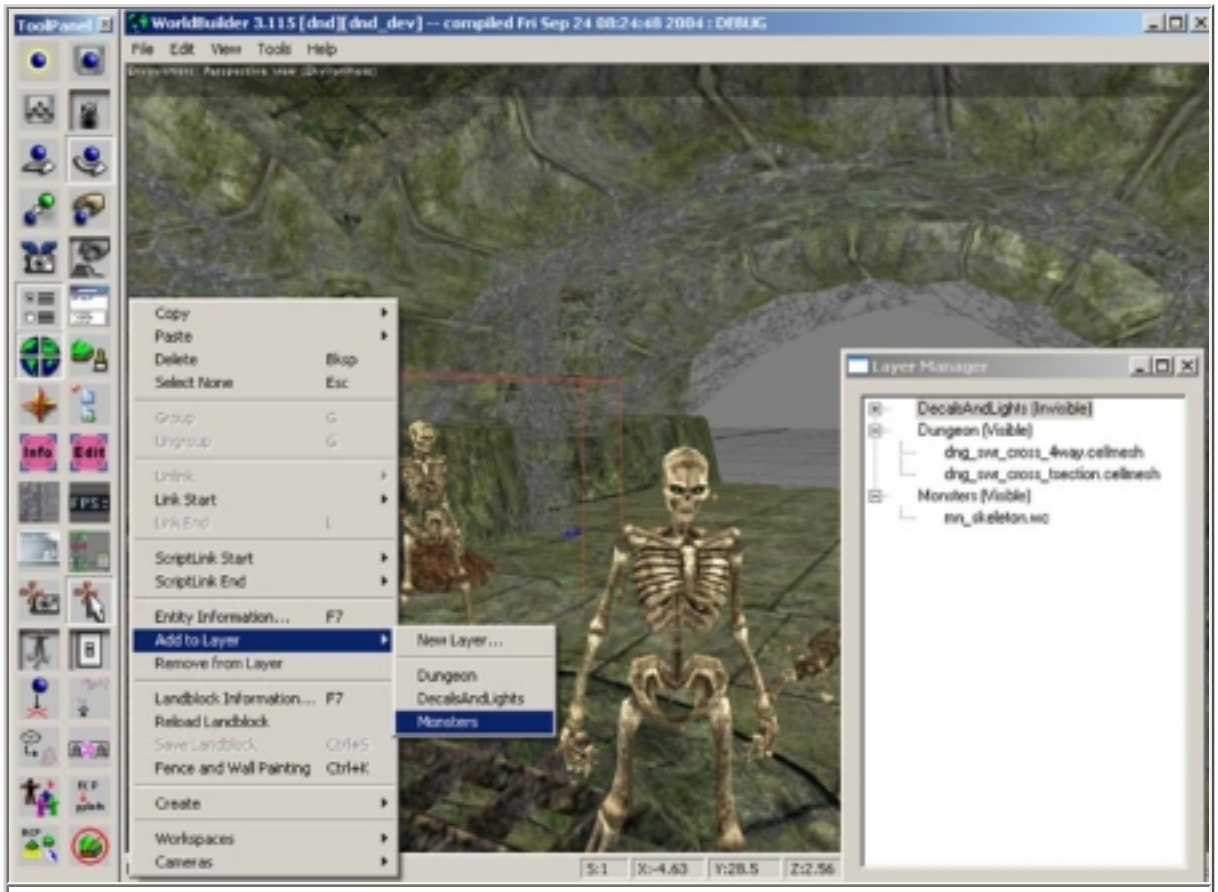
of distinct steps needed to complete the process. When our engineers get interrupted to fix a problem or re-tasked to a high priority feature, they know that there is a document available that will quickly reset their frame of mind.

Another hot topic is the "living document" debate. Are your design documents updated as time goes by? Or are the documents set in stone? Our high level design document is used as the "set in stone" version; the requirements and design ought to be good enough that the document doesn't need revision during the implementation phase. Additionally, the users and the engineers have agreed on everything in that document, so we all have a common understanding of what is going to be provided. The detailed design is used as the "change over time" version; since it's only for engineering use, the engineer may update it as necessary modifications are encountered during implementation. A good detailed design has easily saved a third of the estimated development time; never mind the headaches that it helps avoid.

Prototype and Implementation

Now that there are solid requirements and a detailed design document, implementing and prototyping the tool should be a piece of cake, right? Well, yes and no. It is true that by following your detailed design, you could implement the functionality desired. Coding is generally straightforward - there is a problem to solve and a variety of ways to do so. There-in lies the problem; given a hundred ways to code a function, a hundred different developers will code it *two hundred different* ways. How do you provide some amount of consistency in your tool software? Turbine does so with its coding standards and peer-reviews.

"Coding standards" are a well defined set of rules that govern how code is to be written at your company. Naming conventions for variables and members of classes, tab/space indentation schemes, and bracket and brace placement are typical items found in a coding standards document. These coding standard items do increase the readability of the code; an important factor with a large team writing an MMO. However, a coding standard can and should provide as much information about the software writing phase as possible. Some questions you should ask: Does your coding standard have a common copyright blurb set into every code file? If so, what is it? Are there library, file, and function naming conventions? Should your header files conform to a document generation tool such as Doxygen? It is hard to believe that some companies have no standard method for writing software. Coding standards are not only for ISO-9000 or Department of Defense contractors; they play a large part in the development of any software project. If your company plans to license your engine at some point, having and adhering to a strict set of coding standards will provide your customers with a clean, consistent, and uniform API and codebase.



Turbine's LayerManager

Peer code reviews ensure that multiple sets of eyes (and presumably brains) are looking at every piece of code. The formality of the review process is up to you. At Turbine, we generally have another engineer review code changes at the author's desk prior to checking the code into revision control. Occasionally, the change affects multiple systems and several reviewers are solicited. Often, an engineer will spot his or her own flaws while explaining the code changes to someone else. Because the code is being reviewed at the author's desk, criticism can be given constructively; code reviews held in a conference room with a sheaf of printouts can feel combative to the author. The list of changes spotted by the reviewer are noted either inline or in a separate file, then addressed prior to check-in. The peer-review process has an added benefit: it encourages knowledge sharing; more than one person has visibility into any system we write.

Testing

Tools development has historically been very bad about software testing. There's no time, no budget, and no personnel. We end up pushing out untested tools on our users, who in turn are frustrated with buggy code. This approach ultimately costs us more time in fixing those bugs. Testing software tools takes two forms: white box testing and black box testing.

White box testing is done by the developer, stepping through the code pathways and making sure things are coded correctly. It's called "white box" because the developer can

see inside the box (his code). Black box testing is performed by the Quality Assurance Team; given a set of inputs they expect a set of matching output. They don't have any idea what's inside the box. Your software process should include both sets of testing.

Your developers should at least be testing their own code. White box testing is pretty specific to the functionality being tested. At a minimum, the developer should provide complete code coverage. This means that the developer should test every line of code they write: all of their if/switch statements, successful code path, and error cases. This can often be time-consuming, testing all of the code pathways, but it is the only way to guarantee that the module or function is going to execute correctly under all circumstances. Prior to checking in code, completing a peer-review of the changes (plus a demonstration where applicable) with a fellow developer will make sure that all requirements are addressed, reduce errors, and increase adherence to the coding standards.

Your software process will directly affect how easy it is for a QA team to complete black box testing of your tools. The requirements specification lets the QA team know exactly what changed. The "use cases" in the high level design indicate how the new functionality can be tested. Using those two documents, the QA team can write a test plan for the functionality. Using the test plan and the software tool, bugs or unexpected results are added to a bug databases for fixing and tracking. Additional daily testing can be performed by the QA team, using a functional overview test that exercises the basics of the tool. Turbine now has a dedicated QA department to perform the black box testing. It is true that they are focused on testing the game clients and filing bugs in our internal database, but we are moving forward with plans to have them begin testing the tools as well.

One idea I want to implement is self-testing software tools. By using a command line parameter or having a "Test" menu item, we could allow each software tool to complete a self-diagnostic. Perhaps the tool could create an object in the world, move, rotate, and scale it, delete it, and undo the deletion. Such a simple test case would easily exercise fifty percent of the most common tasks inside our world-building tool. Extending the test cases for each particular tool would give the QA department a single entry point in the tool for daily functional testing, and simplify the burden of testing our tools.

Polishing

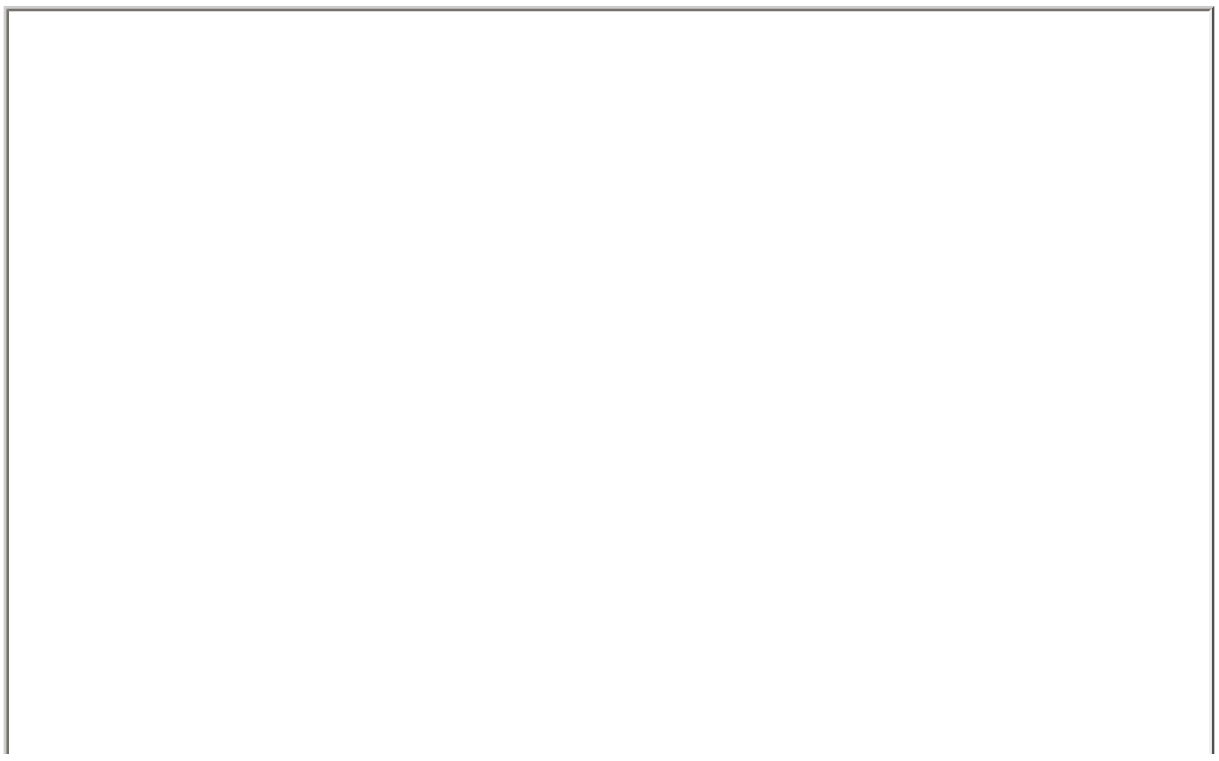
Great! The code has been written and you managed to find some time to do thorough testing; the tools have been distributed. All the users are happy, right? The polish phase is the time to make sure that everything is easy to use, runs fast, and performs well. One of the best ways to do this is to have an internal post-mortem. This may be as formal as gathering everyone in a conference room and airing grievances about the tools; it may be as informal as dropping by the user's desks and asking them what their biggest impediment to productivity is.

Turbine conducted a full internal tools post-mortem towards the end of the *AC2: Fallen Kings* product development cycle. By doing the post-mortem prior to the game going live, we got a glimpse at the problems the designers would be facing during the monthly props. As a result, we created a brand new tool that simplified the generation and editing of our string tables (localizable versions of the game text), as well as pointed out the need to rewrite portions of the basic AC2 text controls.

Using your own tools to complete a software task is like taking a dose of your own medicine. You can receive good feedback by performing usability testing; pairing users and the developer after the tool or added functionality is complete. Tacking on the pressure of working with real game assets alongside a designer, the developer quickly sees what improvements can be made. During polishing, we add features for the "expert user." The two most common additions are keyboard shortcuts and toolbar icons for frequently used commands. Turbine has also implemented a timing viewer that allows us to take a snapshot of the time taken inside various subroutines and displays the results in a tree view. This tool has helped identify times where excessive loading and processing takes place, and allowed us to make enhancements.

Iterate

Finally, are we done? We've covered all five steps of the software process, but we aren't quite done yet. Now that you've completed the process, it's time for the sixth, hidden step: iteration. Iteration is especially vital for MMO tool development. Fifty percent of your Live team will probably be content designers, who will be using your software tools every day for years to come. In our experience, the Live team budget and developer staff does not take into account tools development. This leaves the Live team with whatever was completed by the time the game goes public.



Dungeons and Dragons Online

At the end of *AC2*'s development, the major "tools to-do list" contained more than 75 pending feature requests. Fortunately, we were able to spend another three months on some of those issues following the game's release. However, there are still problems that haunt us to this day. Our engine changes significantly between projects. Consequently, the tools for *Middle-Earth Online* and *Dungeons & Dragons Online* are considerably different from those in *AC2*. We attempt to back-port changes from the latest engine development tree into other game trees whenever possible, but sometimes the underlying engine changes (and budgets) preclude the ability to do so.

A small example of our software process may help illustrate the various phases discussed above. This example was taken from a recently requested feature for our world building tool. The content designers wanted functionality similar to Maya or Photoshop, such that any selected entity could be assigned to a visibility layer and the visibility of the layer could be toggled. Entities and layers made invisible in the world building tool would not be rendered by the engine. A small section of the requirements document follows:

> Requirement 1: A new menu option "Add to Layer ->" will be added to the right-click menu, which will provide a method for adding entities that are selected in the various workspaces to a layer. Entities added to a layer in this manner will assume the visibility status of the destination layer.
>
> Requirement 2: The right click menu will contain an option for "New Layer..."

where the user will be queried for a new layer name.

Requirement 3: The right click menu will contain a programmatically generated list of layers.

Requirement 4: A "Remove from Layer" menu option will remove the entity from a layer, if it is currently assigned to one.

Requirement 5: When one or more entities are selected and the right click menu option "Add to Layer" is selected, the entities will assume the visibility status of the target layer.

Note that each requirement above consists of a single, clear, testable statement. This focuses the reader on one item at a time as well as making testing the requirements easier. At this point, the requirements are distributed, signed off on, and scheduled. This particular task involved creating a new menu and dialog box, as well as a few supporting classes to track the visibility states. The task was further complicated by another requirement to save the visibility state in our data files and restore it when the entities were later reloaded.

The next step was to create "use cases" for these requirements in a high level design document:

Use Case 1: The user has selected some entities in the 3D workspace and clicked the right mouse button. The right click menu will contain the "Add New Layer..." menu option at the top of the menu. A list of menu options will show for "Add to LayerName" where "LayerName" is each layer from the list of previously created layer names. Finally, the "Remove from Layer" menu option will be displayed. (Reqs. 1, 2, 3, 4)

Use Case 2: The user has selected some entities in the 3D workspace, clicked the right mouse button, and selected the "Add to LayerA" menu item. If LayerA is visible, each of the selected entities will toggle their visibility to "visible." If LayerA is invisible, each of the selected entities will toggle their visibility to "invisible." LayerA will now own the visibility state of each selected entity. The VisibilityLayer dialog box will be updated to show that "LayerA" contains each entity name as a child node in the tree view. (Req. 5)

The use cases contain more information than the requirements. It is still possible to map the requirements back to the design: the first use case contains the first four requirements, while the second use case contains the fifth requirement. Numbering the requirements is not required, but it can help identify if everything has been addressed.

The use cases are broken down in the detailed design document to the class/function/module level:

Use Case 1: The user has selected some entities in the 3D workspace and clicked the right mouse button.
- All functionality will take place in Wb3DWorkspace::BuildMenu.
- Determine if any entities are selected. If not, skip generating the menu.
- Add a menu option "Add New Layer...".
- Get the list of VisibilityLayers (new class) from the LayerManager (new class).
- For each VisibilityLayer, retrieve the LayerName and add a menu option "Add to LayerName"
- Add a menu option "Remove from Layer"

From the detailed design, we begin to see the classes, functions, and member variables that will ultimately be created in code. One small section of code generated from Use Case 1 illustrates some of the coding standards we use at Turbine.

```
bool
Wb3DWorkspace::BuildMenu( const bool& i_bSelection,
                          Menu* io_pMenu )
{
  // if the LayerManager hasn't initialized, we cannot
  // build a menu
  if( !m_pLayerManager )
  {
    return false;
  }

  // if the menu is invalid, we can't add to it
  if( !io_pMenu )
  {
    return false;
  }

  // if nothing is selected, we don't need to add items to
  // the menu.
  // NOTE: this case is not an error!
  if( !i_bSelection )
  {
    return true;
  }

  // add the "Add to New Layer" menu item
  io_pMenu->AddMenuItem( ADD_TO_NEW_LAYER_MENU_STRING,
                         ID_ADD_TO_LAYER );
```

```
  // add all of the existing layers as menu items
  const SmartArray< VisibilityLayer* >& arrayLayers =
        m_pLayerManager->GetLayers();
  if( arrayLayers.count() > 0 )
  {
    int il;
    for( il = 0; il < arrayLayers.count(); ++il )
    {
      VisibilityLayer* pLayer = arrayLayers[ il ];
      if( pLayer )
      {
        io_pMenu->AddMenuItem( TEXT( "Add To %s" ),
                               pLayer->GetNameString(),
                               ID_ADD_TO_LAYER + il );
      }
    }
  }

  // add the "Remove from Layer" menu item
  io_pMenu->AddMenuItem( REMOVE_FROM_LAYER_STRING,
                         ID_REMOVE_FROM_LAYER );
  return true;
}
```

Case 1

White box testing of the above code is completed in the debugger, setting a breakpoint at the first line of the function and adjusting the input parameters, as well as the return values from called functions. Some code, such as assertions, has been removed for brevity. Black box testing involved several test cases: right clicking with entities selected and unselected and visibility layers created and absent. The resulting menu was visually inspected to verify that the menu items were only displayed when there were entities selected, and that visibility layer menu items only appeared when there were existing layers.

During the polish phase, we added a few features to the LayerManager: dragging and dropping was enabled for entities between VisibilityLayers and doubleclicking a VisibilityLayer name in the LayerDialog caused the layer and all entities assigned to the layer to toggle their visibility status. Future requests and iteration will likely reveal more improvements for the users. While the previous example is understandably small due to space considerations, it reasonably displays the level of effort and detail used in Turbine's software development process for our tools.

Massively multiplayer online game creation is more than getting a box onto a store shelf. You must think about the future. Similarly, your software development effort is not solely

about the game, it encompasses the tools that the Live team will use to support the game in the future. Ensure the future operation (and income) of your MMO with well designed tools that make content creation easy for your designers. As the saying goes, "An ounce of prevention is worth a pound of cure." Catching a problem early in your tool's design phase will cost significantly less if fixed at that time, rather than waiting and discovering a bug after the tool has been released to the users. A sound software methodology that is iterated on from requirements through polish is crucial not only to game development but to software tool development as well. Is your software process working for you?