Gama Network Presents:

Structuring Key Design Elements

By Erik Bethke
Gamasutra
April 11, 2003

URL: http://www.gamasutra.com/features/20030411/bethke_01.shtml

All games start as an idea, something like "Wouldn't it be cool to be a space marine and blow up zombies on Phobos" or "Wouldn't it be cool to be a pilot in a starfighter involved in an epic struggle to overcome the oppression of a star empire gone bad" or "Wouldn't it be cool to drive modified street cars on Tokyo streets at night." These idea sparks are often the source of long conversations between developers late into the night at the studio. Another potential starting point for a game is a licensed property; i.e., "make a RPG/RTS/action game using XXX license." (Fans may want to play that license specifically. Major licenses include Star Trek, Star Wars, D&D, WWE, Lord of the Rings, and Harry Potter.)

This article discusses how to turn the structure that your business context and your game ideas provide into a game concept worthy of fleshing out into a game design document.

Business Context Shapes Design, Or Does Design Shape The Business Context?

First of all, I am not asserting that having your business context in hand will act as a magical tool that will turn any game idea into a well-thought-out game concept. It is only an important aid to assess the requirements that your game idea is implying. Some game ideas (such as the faithful recreation of Middle Earth where the whole world is modeled with strong AI, 3D graphics capable of great indoor and terrain rendering, where an unlimited number of players can join in together on both sides of epic conflict between good and evil) cannot be reconciled with the business parameters of two artists and a programmer looking to break into the industry, who have six months of living expenses available to them on their collective credit cards. That Middle Earth concept is an example of a game that will dictate the business parameters. If we take the business parameters of two artists and a programmer, they might want to recreate an arcade classic on the Nintendo Game Boy Color or Advance, use it to secure their first deal, and then move on to more ambitious projects.

For many game projects there is a middle ground where the business parameters and the game idea go back and forth and refine each other. Perhaps the developer pitches a massively multiplayer game to a publisher who is wary of the costs and risks behind massively multiplayer. From these talks it is quite possible the developer will end up creating a game that exploits a
license the publisher has rights to and features a much more modest multiplayer feature set. This is not an acceptance of a mediocre plan; rather it is a mature development of the idea into a viable concept. A viable concept is a game that people with capital believe will be seen through to completion, with a high probability of favorable reception in the market to overcome the inherit risk in game making.

**Reconcile the Business Context and Game Idea Early**

This process of refining the game idea and business context is the earliest stage of a game project. All projects reconcile their business contexts and the game idea at some point. Tragically, for too many projects this reconciliation only occurs after the project manifests itself by underperforming, usually by missing milestone dates. Some projects have a painful reassessment where senior management allocates more funds and grins and bears it. For other projects, senior management interprets this late reconciliation as an unpleasant surprise presented by an immature development team and consequently cancels the project.

Allocating more funds and time to a project is a common occurrence, and because it is commonplace, too many developers think it does no harm to themselves and no significant harm to the publisher. That is fallacy; when a publisher is forced to spend additional dollars and push back the release of the title, there are many negative impacts.

First of all, the publisher must extend additional money to the developer. This is an obvious point, but it means that these funds are unavailable towards the development of another title with another developer or (worse for this title) funds may be drawn from the marketing budget to pay for this overage.

The second impact is that the publisher has to delay when they will be able to start recouping their investment and see a profit that they can put to work in future games.

The third problem is that the marketing effort is deflated as the awareness for the game is now ill-timed, and it will be difficult for the game cover that marketing was able to secure for your game last quarter to have real value 18 months later. Right or wrong, the developer is the vendor and the publisher is the customer, and the adage that the customer is always right holds firm in this case, with the developer being tarnished by the reputation of poor estimating capability.

Another reason to avoid going back for extra money and time from your publisher is that the business deal will never improve. A loss of royalty points is common; sometimes you will see a shifting of intellectual property rights. In the extreme sometimes the developer agrees to an assignment of equity in the project to the publisher. In the case of shifting equity to the publisher, the developer is strongly advised to get full value for that equity; no matter how small an equity stake the publisher takes, it will make all other publishers avoid doing business with the compromised developer for fear of a conflict of interest and confidentiality concerns.
The developer is also losing time by going over his time budget, and spending more time on a project with the business deal worsening is not a good goal.

The final reason to avoid a late reconciliation of the business context and game idea is to prevent team members from becoming disillusioned and moving on to another company.

At Taldren we have released *Starfleet Command*, *Starfleet Command: Gold*, *Starfleet Command: Neutral Zone*, *Starfleet Command 2: Empires at War*, and *Starfleet Command: Orion Pirates* in less than two years. At the same time we gathered more fans and have always produced a profit for our publisher. Many of our employees are loyal to Taldren because of the steady pace of release; they know their work will be released and not wasted.

**The Effects of a Slipped Game**

1. Less working capital for the publisher.
2. The total advance is tied up longer than expected.
3. Marketing dollars are often wasted as the hype bugle is blown too soon.
4. The developer's reputation almost always suffers.
5. The business deal never improves for the developer.
6. The developer loses the opportunity to work on other titles.
7. Team members are in danger of becoming disillusioned, and the team may suffer uncomfortable turnover.

Ion Storm has to be the most infamous example of the consequences for late reconciliation of the business context and the game idea. Ion Storm was founded around John Romero, who is credited with the design of *Doom*—perhaps the greatest PC game ever. The UK-based Eidos was flush with cash, and John Romero left id just as *Quake* was entering its final stages towards release. Eidos needed to put the surplus capital from the *Tomb Raider* series to work, as all businesses must do. Tomb Raider was so successful that Eidos needed to get into a number of games, but established top developers were already booked, so Eidos would need to go with a less established development house. The idea of taking advantage of the designer behind *Doom* and creating a new development house is not a bad idea; in fact it is a good idea. Experience, a built-in fan base, and a great story for the media would create an environment that would be conducive to game development, one would think.

Ion Storm was founded with the vision statement that design is king. Even this is not a bad idea; treated properly this would mean that Ion Storm would capitalize on its core strength—game design embodied by John Romero -- and take advantage of existing game engines. Looking at how Ion Storm interpreted their vision statement would reveal where Eidos made their mistake. Ion Storm used the vision statement, design is king, to treat game development as a pure art form and lost respect for a strong development process. Ion Storm's marquee project *Daikatana* suffered all of the ills described above. Whole engine retooling caused massive delays and required Eidos to double the already overgenerous advance of $13 million to $26 million to keep Ion Storm's three projects rolling.

*Daikatana* did not just lose face in the game press, it became the material for much derision, and even the local Texas newspapers saw the poor management at Ion Storm as a good story for a series of columns. Ion Storm not only suffered crippling turnover, but some employees helped feed the negative press storm by leaking to the press ugly internal email. John Romero was
forced to hand over the company to Eidos, and their games shipped to little success. Ion Storm's Dallas office has been closed by Eidos to what amounts to a large write-off of Lara Croft earnings and a reputation for Eidos to overcome. In fact the quieter Ion Storm Austin studio run by Warren Spector, which shipped the critically acclaimed *Deus Ex*, is now looking for a shiny new name to operate under to distance that studio from the ill-fated Dallas studio.

The sad thing is that John Romero really can design games; just play *Doom* any day and you will see how amazing a game it was and still is. And Eidos turned on the cash to set up the game for greatness. It is just heartbreaking, really, to think about the potential of Ion Storm and to see it fall for lack of rigorous development methods.

What can be worse than either pumping more money into a late project or canceling a project? *Shipping it.* It should never be done, but almost every large publisher has shipped a game well before it was finished. I don't mean just with bugs; I mean before critical parts of the game were complete.

*Descent to Undermountain* from Interplay is a classic example of a game that was shipped too early. The idea behind *Descent to Undermountain* was to take advantage of two key assets of Interplay: the *Advanced Dungeons and Dragons* license and the mega-hit *Descent*. Management at Interplay decided it would be a snap to plop down some fantasy environments, characters, and monsters to bash. Management decided the *Descent* game engine would be ready for immediate development into another title. Most publishers do not have a strong technical director available for code review. Yet at the same time many publishers also negotiate the terms of the publishing deals to either own the software engine behind the game or have a license to the software engine. *Descent to Undermountain* was a case where the revenue opportunity was so large as to prevent an objective review of what it would take to get the game done. The original business parameters for this title called for a budget of only six months of four developers' time. No established development house was chosen to do the job; rather an ambitious independent contractor programmer stepped up, and various artists at Interplay contributed to the project. No project manager was allocated. Let me share with you what Gamespot thought of the results of this game after it slipped to three years and *six times the original budget*:

> From Gamespot review of *Descent to Undermountain*:

> But somewhere along the line something went horribly wrong, and now gamers are asking themselves two questions. The first arose merely out of befuddlement: How could the company that produced *Fallout* also be responsible for one of the lousiest games to come down the pike in quite a while? The second, though, addresses a much more serious issue: Why did Interplay ship the thing when it wasn't even close to being the sort of cutting-edge product the hype machine had led us to believe it would be? ...There's probably no way to learn the answer to the first question, but-thanks to some very frank members of the *Descent to Undermountain* team-the answer to the second is now common knowledge. The game went out when it was scheduled to go out (in time for a Christmas release) even though it wasn't ready. That's not just me speculating; that's precisely what a member of the DTU team stated in a recent post on Usenet.

When a project is three years in the making and six times the original budget, there is tremendous pressure to just ship the game. At the time, Interplay was receiving a huge amount
of attention for *Descent to Undermountain*; everyone wanted a truly 3D dungeon romp. (*Dungeon Siege*, the first really 3D dungeon romping game, and BioWare's *Neverwinter Nights*, which is a more detailed 3D implementation of D&D, weren't released until 2002.) Interplay thought at the time that with all the hype, maybe, just maybe, the early sales in the first few weeks would be large enough to recoup a significant portion of the costs. It was also Christmas time when 40 percent or more of our sales as an industry happen. Interplay had three choices:

1. Ship it now.
2. Cancel the project altogether. (Remember lost money really is lost, and it is best not to chase it.)
3. Find a real AAA development house and start over with a new large budget and two years more of development time. (Really the same thing as canceling the project.)

Unfortunately for Interplay at the time, canceling the project or starting over with a new developer appeared to be more expensive than shipping the title. Let us see what Gamespot thought of this decision to ship the game:

**From Gamespot review of *Descent to Undermountain***:

*The lesson to be learned should be obvious: If you’re gonna ride the hype machine, you'd better deliver the goods. Sadly, DTU doesn't even come close—and the worst part is that sometime over the next year or so we'll probably see this same story played out all over again.*

*So what have we learned today? That pushing a product out the door before it's ready makes loyal customers angry; that game developers should keep at least one eye on what's going on in terms of technology when working on a new game; and that if you buy *Descent to Undermountain* after reading this, you get what you deserve.*

*Descent to Undermountain* shipped in a condition that was far below the industry standards of the time, *Diablo* and *Quake II*. The hype behind this game also crushed it. It is just possible that if Interplay had developed this title quietly, hard-core fans of *AD&D* and/or *Descent* might have bought 20,000 copies and been patient for a patch or two. I am not saying this is a great idea, but it is better than a hype storm. This is a poor way of doing business; the game industry shows time and time again that the mega-hits are just games that offer straightforward gameplay with strong production values. Wacky or niche games or poor craftsmanship are not rewarded. Just make a few quality titles and you will spend a lot less money in development, and your individual titles will have more capital to work with.

*Descent to Undermountain* was a perfect case where the game idea and the business parameters were in conflict. If Interplay wanted a title in six months and had only a modest budget to accomplish it with, then Interplay should have commissioned the developers of *Descent*, Parallax, to create a cool expansion pack for *Descent* and they should have contented themselves with the sales of an expansion pack. Perhaps it was perceived that with *Descent II* already in development at the time, it would have been competing for sales. The other option was for Interplay to allocate the funds they were to later plow into *Stonekeep II* and hire a top developer to create a 3D dungeon romp of quality. *Stonekeep II* would later go into production for five years and then be
Methods and the Unified Development Process

Microsoft, the most successful software development organization on the planet, sells a lot of games. Microsoft is perhaps best known for its Flight Simulator franchise, but MS now owns Ensemble (Age of Empires franchise), Bungie (Halo and Myth, formerly the premier Macintosh development house), FASA Interactive, and Digital Anvil (the former Chris Roberts company working on Freelancer), as well as being the publisher for a host of externally developed titles such as Dungeon Siege. Microsoft is a large organization with many layers of development procedures that other publishers do not employ. The first thing Microsoft does when evaluating a developer is to send a small team of game development leads comprised of production, design, programming, and art to evaluate the strength of the team. A large part of this evaluation is to also evaluate the developer's methods to determine if they are compatible with Microsoft's and if these methods give Microsoft confidence that the developer has thought through their project and will deliver a great game, on budget and on time. Development methods must be good things judging by Microsoft's success.

What Is a Development Method?

method noun-A means or manner of procedure, especially a regular and systematic way of accomplishing something.

We do want systematic game development; this whole book is dedicated to the presentation of various game development methods. Systematic and repeatable methods allow us to retain what worked and improve upon what did not work well. The alternative to using a method is employing ad hoc techniques over and over again and being successful only by good fortune. I rather like to make my own luck, thank you very much. The first method we need to nail down is how to reconcile your game idea and business parameters. I advocate using a comfortable subset of the Unified Software Development Process developed by the three amigos Ivar Jacobson, Grady Booch, and James Rumbaugh.

Why Use the Unified Software Development Process?

The simple reason is that the Unified Process is quickly becoming the software industry standard. The Unified Process has a long legacy dating back to at least 1967; at this time Ivar Jacobson worked for the telecom giant Ericsson. Jacobson had a radical idea for the design of the next generation telephone switching equipment at the time, a method we would now call component-based development. For this project Ericsson modeled the whole switch system and subsystems as interconnected blocks. The relationships between these blocks was then articulated and revised. The dynamic processes of the switch were identified and modeled. Every message passing back and forth from each object was included in this model. This software architecture and object message compilation was probably the best technical design document of the time. This was a radical concept because software customers at the time were not accustomed to seeing a blueprint of the software before the software engineering began. This method was not chosen on a whim; rather it met the demand that the software be robust enough for the telephone switching equipment to remain operating while receiving upgrades and patches to the software components of the switch in real-time.
I will skip the middle part of the history behind the Unified Process; the point is that 35 years ago a repeatable method of creating great software was developed, and despite this, most software organizations have weak methodology.

The Unified Modeling Language is the standardized text and visual language for the articulation of software design supporting the Unified Process. Beyond the development of Ivar Jacobson, Grady Booch, and James Rumbaugh, UML enjoyed broad support and major companies such as IBM, HP, and Microsoft joined in the development and standardization of UML.

**Requirements Capture**

The purpose of a software development process is to take the user's requirements and transform them into a functional software system. That transform stage is what we game developers are doing when we make games. We take the vision of the gameplay-how it should play-and turn that into a finished game.

**JARGON:** Requirements capture- articulating the requirements the functional software must satisfy, such as to be fun or to run at 30 frames per second.

What is the first step in the development process? Figuring out what we are supposed to do. There is a neat formalized term for this: requirements capture. Requirements capture is something you have already started. Your business constraints are some of the requirements the software must satisfy. How do we methodically discover the rest of our game's requirements? The short answer is that there is no quick, magical method to sit down and write up in a single sitting all of the requirements your game must fulfill. Wait, don't go away, I am still going to show you how to do it; it just involves several iterative steps.

**Use Cases**

First, if you have not already done this, write down your game idea on a single sheet of paper. Write two or three sentences that describe your game in the center of the piece of paper. Now in no particular order write down the major functionality of the game in an outward, radial manner from the game idea in the center. The larger, chunkier aspects of the game should be close to the center and the detailed ideas farther away. For example if you are designing a role-playing game, you have characters; write that down. Characters have stats; write that down. Characters have names; creating the characters' names is a feature. What you are doing is brainstorming the gross feature set of your game. This particular method of putting the game idea down at the center of the page is good to get you started if you have not put a lot of effort into your game design document yet. The immediate goal is to identify all of the core activities the player can perform in your game. Each of these core activities is composed of many individual actions the
player performs. Each of these actions is called a use case in the Unified Modeling Language.

**JARGON: Use case**—an interaction between an actor and the software system. A fully articulated use case is composed of both text describing a sequence of actions and a graphical diagram showing the relationship of this particular use case with others in the system.

Collecting these use cases and writing them down will drive our process to identify the requirements of our software. The software requirements will then help us develop the architecture for our software. The use cases represent function, and the architecture represents form. The Unified Process is called use case driven because it is the effort to capture our use cases that drives the development. All of our future efforts in the construction of our software are to further the realization of these use cases into a functioning software system. Now, what exactly does a use case look like?
It turns out one of the fundamental tenants of UML is that the language shall be extensible, flexible, and ultimately serve only to aid the process of distilling and communicating the system requirements. This ATM transaction diagram uses only three UML symbols: the oval use case, the stick figure actor, and the relationship line. The stick figure is called an actor. Actors represented by a stick figure are most often users of your software, or players of your game, who are interacting with the game. It is better to use the abstract term "actor" so you will see all of the external users of your game system such as the single-player player, the multiplayer player, the system administrator, and the database server of your online component. After identifying your actors, the use cases will flow rapidly. The use cases are the unique interactions between the actors and the software system (game). The use cases are represented by a simple oval with an active verb phrase such as "withdraw cash" or "analyze risk."

**JARGON:** An actor is a user, either human or another external system, that is interacting with the system under analysis.

**JARGON:** A relationship is a line drawn between actors and use cases, sometimes with extra notation that further describes the type of relationship, such as <<extends>> and <<uses>>.

The level of articulated rigor in a diagram should be reasonably proportional to your needs. For example, if it is important to describe the relationship line in better detail, use a one-word descriptor between the less than and greater than symbols. Common examples are the relationship descriptor of <<extends>>, <<uses>>, where extends would communicate that a particular use case is really a special case of a simpler base use case, and uses would indicate that a particular use case employs another use case as part of its action.
**Player Input**

Insert coin  
Push coin return  
Choose single player or multiplayer  
Move up, down, left, or right

The use cases of *Pac-Man* related to player input.  
(Click image to enlarge.)

**Game Object Interaction**

Wall collision  
Eat dot  
Eat power-up dot  
Eat fruit  
Eat ghost, send ghost to center of box  
Pac-Man dies

The game object interaction use cases of *Pac-Man*.  
(Click image to enlarge.)
We can also take a higher-level view of *Pac-Man* and combine these low-level use case diagrams into a generalized use case view of the software package as a whole. See the diagram to the right.

There are typically only four roles: a programmer for the 2D display system, a programmer for the game mechanics, an artist, and some audio. This of course is a very small game, and a solid *Pac-Man* clone could happen inside a weekend for a two- to four-person team.

This process of understanding how something else was put together has a fancy name: reverse engineering. I highly encourage you to perform some reverse engineering on other games that you are familiar with. We continue with some sketches from other games.

**Case Studies**

It is now time to apply these tools to modern games that are of greater complexity than *Pac-Man*. Each of the following two games, *Diablo* and *Gran Turismo 3*, has enjoyed legendary marketplace success, and each has spawned a lucrative franchise of sequels, expansions, and licensed products. Is there a common thread between these games? Did the developers in each case just get lucky, or were the developers just extraordinarily brilliant? I honestly do not know how much luck was involved, but someone with a lot of intelligence, skill, and time honed these two game concepts into production plans that have succeeded far beyond the industry standard. I can show you the elegance in the design of these games, illustrating how, looking back, these were mega-hits from their conception.

**Case Study 1: Diablo**

*Diablo* is a computer role-playing game for the PC developed by Blizzard North, originally an independent developer of another name bought by Blizzard during the development of *Diablo*. *Diablo* featured the killing of hordes of monsters like skeletons, wandering around in a dungeon, gathering gold, and collecting magic items all in the quest to vanquish ultimate evil—all...
straightforward fun stuff. The key concept behind *Diablo* was to make the user interface priority #1, not the story, not the size of the game, not the number of different character types, not customized character appearance, not a rich role-playing game mechanics set; no, the focus was the user interface. Indeed, the mouse controls were a stunning left-click on monsters to attack, left-click on chests to open, and right-click to cast a spell. The interface itself was appealing to look at with large glass spheres that held blue and red liquids representing remaining life and mana (energy to cast spells).

Shortly I will more carefully break down the use cases of *Diablo*; but there is a tremendous amount of courage and insight behind the user interface design of *Diablo*. In the summer of 1995 I was up late one night with a bunch of other game developers talking about games we could make. I remember we suggested just a simple variant of *Gauntlet*, the arcade classic where you just went around bashing monsters, collecting gold, and powering up. I remember how we all laughed at the time and said there was no way it could be viable. No publisher would see the game as feature-rich enough to fund. Perhaps as a bit of forgotten shareware, but no way it could be a commercial game. At that time RPGs such as *Bethesda’s Elder Scrolls* series were vast worlds with hundreds of NPCs, dozen of cities, hundreds of locations, actual weather, and time of day. Imagine making a game that left out all of these features and just concentrated on a tight interface and high production values—that was *Diablo*.

### Use Cases of Diablo

*Diablo* is a simple game, a polished game with strong production values such as superb voice-overs and movies, but we will see that Diablo is a simple game behind the features. I will cover the major features and elements of the game; I do not propose to create an exhaustive reverse design document.

<table>
<thead>
<tr>
<th><strong>Display System</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Terrain: Draw floors.</td>
</tr>
<tr>
<td>Terrain: Draw isometric walls.</td>
</tr>
<tr>
<td>Terrain: Color cycling special effects for water and lava (tiles do not animate).</td>
</tr>
<tr>
<td>Terrain: Ghost walls when a character is located behind the wall.</td>
</tr>
<tr>
<td>Characters: Render and animate characters (2D sprites composed from 3D rendered models).</td>
</tr>
<tr>
<td>Game Objects: Colored outlines for interactive objects such as treasure chests, magical rings, monsters, and non-player characters in the town center.</td>
</tr>
<tr>
<td>Spell Effects: Display any one of a couple of dozen spell effects with dazzling animations and cool sound effects.</td>
</tr>
<tr>
<td>Menus: Display menu choices.</td>
</tr>
<tr>
<td>Movies: Display the intro and exit movies to the player.</td>
</tr>
<tr>
<td>Audio: Hear sound effects.</td>
</tr>
<tr>
<td>Audio: Hear music.</td>
</tr>
</tbody>
</table>
Audio: Hear voice-overs.

The view related use cases of *Diablo*.
(Click image to enlarge.)

**Game Object Interaction**

Move Player Character: Left-click to move the player character.

Left-click Object Interaction: Interpret the left-click on an object automatically by object type to mean open a chest, attack an enemy, or move the player to a location as above.

Load Level: When the player directs their character into special trigger areas on a map level, load the target map level.

Right-click Object Interaction: If the character has a spell bound to their ranged action, cast a spell at this location on the map or on this character (this could be either an offensive spell on an enemy or an aid spell on an ally). Otherwise if this character has a bow, fire an arrow at the character indicated.

The game object interaction use cases of *Diablo*.
(Click image to enlarge.)

**Character Management**

Name Character: Small feature for user customization to allow the player to bond with their character.

View Character Stats: View attributes, health, experience points.

Allocate Character Attribute Growth Points: When the character achieves the next experience allow the player to choose where they want the growth points to be allocated, choosing from strength, dexterity, intelligence, and constitution.

Inventory: Display the character's inventory in a "paper-doll" fashion with sockets for the backpack, belt, helmet, hands, pants, boots, and tunic locations.

Inventory: Allow the player to shuffle objects about in their backpack to "make room" for new treasure and to abandon lesser treasure in favor of higher prized treasure. Validate the placement of inventory items based on their type. For example, healing potions can be carried in the backpack or in the belt pouch but not in the helmet slot.

The character transaction use cases of *Diablo*.
(Click image to enlarge.)
Quick Analysis of the Use Cases of Diablo

Looking over the use cases of Diablo you will notice that I have partitioned Diablo into three subsystems: Display System, Game Objects, and Character Management. Below is a short discussion of these systems.

The display system is just a 2D isometric engine that is capable of rendering animating 2D sprites (quite probably used for both the characters and the spell effects). This graphics technology was hardly groundbreaking in 1997; isometric engines have been around since Q-bert in the arcade. The game also uses a 256-color palette incidentally. There is no question that the graphics in Diablo look strong; the art direction was strong and led to a consistent look that was foreboding and well supported the theme of the game.

Touching on character management for a moment, the display system is called upon to also render menus such as the menus of the town shopkeepers who have stayed behind after the arrival of demonic forces to make a profit selling adventuring gear to the player's character and the inventory, spell, and character management menus. These again are just menus, displaying customized fonts, buttons, icons, and cool negative space textures.

The characters in the game animate well due to the aggressive use of 3D rendering to produce the 2D frames from which to composite the 2D sprites. This technology is not new either; our example Pac-Man uses just a few frames from open mouth to closed mouth to animate our hero, and the Wing Commander series used an array of images (about eight to sixteen individual images) from all angles around the starfighter to produce its "3D" starfighter game. The plan for Diablo was to again use established technology but take it to a quality level never before seen in games by using over 5,000 frames of animation for just the three main protagonist characters. This dedication to visual fidelity represents a lot of confidence in staying with established technology but taking it to a very high level of quality. I know of another game I will not mention by name that became severely distracted with the pursuit of volumetrically projected pixels, known as voxels, for the rendering and animation of their characters. This distraction helped to cripple this title.

The game object interaction system runs the heart of the game. This is a game of hack and slash and loot gathering. The context of this hack-and-slash has something to do with a crystal in
somebody's head, demons from hell, a butcher, and dead townsfolk—plenty of motivation to keep our player character hacking away at the monsters in the game. The game object interaction handles the combat, spell casting, opening doors and chests, triggering traps, and level changes. Notice that my use cases above do not have any detail on how combat, spell casting, or the opening of doors and chests works. Those are detailed use cases that would be covered in the design document; this article is focusing on the key design elements of the game in the effort to be sure we have the correct scope for our game.

My use of UML's use case notation has been purposely slim with the use of just the simple table format of major user interactions and a few diagrams to show the relationship of these interactions with each other.

**Case Study 2: Gran Turismo**

The *Gran Turismo* series for the PlayStation and PlayStation 2 platforms published by Sony is all about racing cars. Every conceivable subgenre of racing has been explored over the years as well as many sequels offering the latest technical wizardry for themes already visited. Racing cars have been a staple of video games since the days of the Atari 2600 with *Night Driver*, where the road and terrain are a solid field of black demarked thoughtfully with some magenta lane markers. Nighttime racing has continued to evolve to *Tokyo Street Racer* on the PS2 and *Project Gotham* on the Xbox. Racing games deliver an experience that almost everyone wants to do—race cars. Some want to race at night, some off road, some want to race taxis, some want to run over pedestrians, but hey, there is a racing game for everyone.

What is it about *Gran Turismo* that makes it a mega-hit? Was it luck? Was it a large budget? Or was there some sort of planning and direction behind Gran Turismo? I am presenting a case for thoughtful planning.

*Gran Turismo’s* *(GT)* vision statement was most likely something like "The best racing simulator on any platform." To back up that vision statement we need to look into what it would mean to be the "best racing simulator." The best is so encompassing in its superlative that Sony set out to dominate all other racing games. Hmm, that is a tall order. The first step is to pick the type of racing Sony would model. In the end, Sony chose to model a variety of racing from raw amateur racing of minivans to world-class events featuring million-dollar racing machines achieving the highest form of automotive engineering.

So, at first glance it would appear that Sony violated the design guideline of focusing on one game and a tight set of features and doing them well. However, if we take a look at how they presented these various classes of racing to the player, we will see that it was a seamless presentation of gameplay from the lowliest of minivans to the Suzuki Escudo.

When you load up the simulation mode of *Gran Turismo* for the first time (it doesn't matter which version), you are given a small amount of credits to purchase your first racecar. Taking a look at the various car manufacturers, the player has only a couple of choices in the beginning of the game. After spending all his cash, the player then sets out to race some beginner races to build up a supply of cash so he can modify his car. The car modification gameplay is the hidden weapon of *Gran Turismo*. Here players can ogle new tires, polished ports, oversized turbos, and a host of other modifications to their car. The exhaust improvement conveniently enough has the highest bang for the buck and will most likely be the first purchase for any player. Here the player bonds...
with his car, and all the cool parts available drive the player to go back to the track and keep racing. This context for the racing is compelling. It is the same inventory/party growth dynamic from a role-playing game like *Diablo* - a most compelling feature.

This racing around a track and modifying the car goes on and on throughout the whole game. What changes are the events, the tracks, the competition, and most importantly, the car the player is racing. *Gran Turismo* features hundreds of cars, dozens of tracks, and scores of events. The events are classified into licenses from Beginner to International A. Players can always find a race and almost always can earn some cash to make forward progress on acquiring new goodies for their car. This car modification meta-game is what ties all of *Gran Turismo* together and presents to the player a world where they can start with a modest real-world car, and through racing, modifications, and licensing they too can be an international racecar driver. This is the brilliant vision behind *Gran Turismo* -- it slowly builds up to the super cars, and all along the way the player is hooked and believes in the world and knows why he is playing this game.

Later in the series *Gran Turismo* added rally racing. This additional mode of racing was also seamlessly integrated into the core game. Indeed, the player's rally racing cars just needed to change the tires to racing slicks and they would often do well in the pavement events. In classic arcade fashion, new tracks would only be revealed to the player after completing a racing series or a licensing program. The rally events in the later *GT* series upheld that tradition with their own set of rally tracks to unveil. The *Gran Turismo* series is the greatest of the racing games because it fully delivered on the gameplay that is central to racing and takes players from knowing nothing about racing cars to being able to carry on an extended conversation about gear ratios and coil-overs.

I justified *Gran Turismo*’s success without ever mentioning that the game has always boasted the most realistic physics model for its racing, the most gorgeous graphics, and a complete aural experience second to none. All of these technical features are of course critically important to an electronic game; however, it is the key features of a game that will lead to success and enable the project to fully realize the efforts of the whole game development team.

**Use Cases of *Gran Turismo***

<table>
<thead>
<tr>
<th>Car Driving Controls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Press the Gas Pedal</td>
</tr>
<tr>
<td>Use the Normal Brakes</td>
</tr>
<tr>
<td>Turn the Car Left or Right</td>
</tr>
<tr>
<td>Shift the Gears Up or Down</td>
</tr>
<tr>
<td>Use the Emergency Brakes</td>
</tr>
</tbody>
</table>

The player input use cases of *Gran Turismo 3.*

(Click image to enlarge.)
Display and Audio System
- Render the Track, Terrain, and Sky
- Render the Cars
- Render the Special Effects
- Play Sound Effect and Music

The display and audio use cases of *Gran Turismo 3*.
(Click image to enlarge.)

Shell Activity Menus
- Access Buy Car
- Access Garage
- Access Wash Car/Oil Change
- Access Race Car
- Access Modify Car
- Access Licensing Tests

The shell menu use cases of *Gran Turismo 3*.
(Click image to enlarge.)

Modify Car
- Browse Major Systems: Engine, Transmission, Aerodynamics, etc.
- Review Individual Item: Read the stats of this item and see how it would look on the car if it is an external add-on or what the change to weight and power would be if it is a performance item.
- Purchase Item: Buy the specified upgrade part.
- Install Item: Have the newly purchased item. This especially makes sense for the purchase of tires; it is useful to the player to be able to choose from a suite of tires.

The modify car use cases of *Gran Turismo 3*.
(Click image to enlarge.)

Quick Analysis of the Use Cases of *Gran Turismo*

Again, this article is not discussing how to complete a detailed design document, so I have only covered the higher-level functions of *Gran Turismo*. But in two areas, driving the car and
modifying the car, I drilled down to the individual interactive activities the player has to play with. Driving the car and modifying the car is the game; everything else is in context of these two activities.

Gran Turismo is successful largely due to a clear vision and plan for the game. It was perfectly designed to capture the largest segment of the market who would enjoy racing games. In fact my father and his best friend went out and purchased PlayStations after playing Gran Turismo 1 at my house and went on to compete with actual cash prizes for virtual driving seasons. These two men in the over-50 demographic were not hard-core gamers; they were mass-market consumers who bought the PlayStation just to play Gran Turismo. That is a true hit.

The Key Design Elements of Your Game

I am sure you are now comfortable with this light introduction to UML use cases. They are hardly more than a table of actions and a simple diagram composed of a stick figure and bubbles of action. Now I want you to think about the interactions of your game and write down its use cases.

The methodical way of discovering your use cases is to focus on the core activity of your game and write down all the things the player does in the core of your game. Work your way outward, writing down the other activities you have planned for your game, such as buying gear, building a house, researching flame throwers, learning a new spell. Keep working outward until you can't think of anything you missed. At this stage we are looking for the major activities, so don't think about how many buttons the save menu will have, just what are the big interactions between the player and the game.

Then sort these activities into groups based on similar functionality as I have done with Diablo and Gran Turismo. Finally sketch out the use case diagram complete with the player actor and your use cases. It is useful to create diagrams for each group of activity. You have now articulated your gameplay in both an easy-to-read text format and graphical format. These use cases will be the basis of refinement for the game design and technical design stages. However, in this article we are looking for key design elements. Examine your groups of activities and look hard for a set of activities that stand out as potentially unnecessary to the core of your game. Are there parts of your game design that are distracting in complexity? Are these parts only fun to a hard-core set of fans? Are these features hidden from the novice player? Can they be cut altogether?
Take a look at your design; are you sure you are only making one game? I think a lot of the projects that slip by years make the mistake of trying to fold more than one game into a single game project. You do not need to make more than one game to be competitive. Just make a small set of features that are inherently fun, make those tight, and take the production values as high as possible. This is how a hit is made.

The Battle of the Counterterrorists Games

There are two games that neatly make the point I am discussing in this article, nailing the right key design elements. These two games are *Rainbow Six* and *Counter-Strike*. Both of these games feature special operations type protagonists working as a team to defeat terrorists and other modern day bad guys. An experienced development team produced one of these games with a full development staff for an established publisher. The other game was developed principally by two fans who have had experience making mods with modest financial backing of a development house.

Both of these games are successes and I would be proud to have been a team member in any capacity on either of these two projects. That being said, *Counter-Strike* clobbered *Rainbow Six*. *Counter-Strike* is the mod produced by a small staff of fans working part-time, while *Rainbow Six* is a full game with many man-years of effort. If game development is so hard, how could these fans have done so well compared to the pros?

While poor technical execution will never make a hit game, the answer to this question lies again in the key design elements of *Counter-Strike* versus *Rainbow Six*.

The Key Design Elements of *Rainbow Six*

*Rainbow Six* was the earlier of the two games; to some degree this can never be a fair comparison, as the *Counter-Strike* mod team had *Rainbow Six* available to experiment with and to refine. *Rainbow Six* was designed for single-player play, and while it did have multiplayer mode, the game was much more playable in its single-player mode. *Rainbow Six* featured an extensive campaign structure where you managed the team members of your elite special forces. This team management would appear to be at first glance quite fun and supportive of the context of playing the missions of *Rainbow Six*, much like *Gran Turismo*, and that might be true. However, the *Rainbow Six* team added another context layer to the game: mission planning. Here the player planned out the mission to such a degree that they could tell their team members when to throw the flash grenades and which doors to break down and which to sneak through. After the planning stage was complete, the game acted somewhat like the blend of a movie and a game experience. The movie experience came in where your AI teammates, whom you gave instructions to prior to mission start, would follow your orders and have whatever success might befall them; the game part was that you still had interactive control over your character.

Are We Playing a Mission or Planning a Mission?

I think the preplanning of the missions is what prevented *Rainbow Six* from taking off to a higher level of success. The problem with such a detailed modeling of the preplanning stage is that it was cumbersome in three ways: First, the player already had context for the missions through the campaign structure and the team management feature sets; second, it was cumbersome due to
the user interface of the preplanning. It was like having to act as some kind of game scripter, programming your teammates. And finally it was cumbersome; each time you died or otherwise failed on your mission, the player would break out of the cool, immersive action of the mission and be forced to calculate new scripting paths for their AI teammates. All of these awkward bits leaked out throughout the game-playing experience, leaving me wondering if the designers of the game ever came to agreement about whether the game was about playing the mission or playing the premission planning.

RAY MUZYKA SPEAKS: I totally agree. I recall being very irritated with how difficult it was to equip your party, choose your party, plan out your party’s actions etc. There was no learning curve; instead you were dumped into an equipping-your-character simulation, which, fundamentally, was not the game I had thought I was purchasing. This created a perception/reality gap for the consumer that made people not want to play the game.

The Key Design Elements of Counter-Strike

Counter-Strike was designed to have only a multiplayer mode; not even a training simulation against bots like Quake III was available. Counter-Strike's brilliance is much like Diablo's in its courage to strip away game features and polish the core game until it is humming with game shine. For years in first-person shooters, when you died you instantly respawned to frag again. This is of course a load of fun, as one could easily spend a few hundred hours blowing away your friends before you get bored. But eventually people did get a little burnt out on straight death match, and a desire for something more manifested itself. These explorations for more came in the way of mods for Quake and Unreal that had different victory conditions for winning such as capture the flag. The team that produced Counter-Strike took the idea of a mod with context to the next level (that, by the way, is an overly worn phrase in the industry, but it sure is handy).

The next level of gameplay in a first-person shooter was to wrap an economy about the fragging of the game through credits one earned by winning missions and getting frags. This economy would enable the player to buy larger and more capable weapons, armor, and grenades, which in turn would enable him to perform even better and potentially get even cooler equipment. This feature combined with the idea of a death where the player had to sit out the rest of the turn really helped to focus the player on the harshness of the Counter-Strike world and put some good tension back into the game. Players would carry their credit balance forward each time the mission was over, and the frag counting would continue. Thus, Counter-Strike was designed in the beginning to be a replacement for the endless multiplayer fragging and instead be a much more compelling way of playing extended multiplayer first-person shooter action. All of this was accomplished by the thinnest of user interfaces, on top of Half-Life's version of the Quake engine.

In my opinion the Counter-Strike team really understood the gameplay experience they wanted to deliver—the most visceral counterterrorist gameplay experience, period. In the case of the Rainbow Six team, I think they were handicapped by the source material from Tom Clancy's Rainbow Six in choosing to model the extensive preplanning stage of a mission. That stage is no doubt realistic and the larger portion of the job in a real counterterrorist mission, but it just gets in the way of having fun hunting terrorists. And we are in the profession of delivering fun, not realism. Realism should only be used to create fun, not detract from it.

Most Popular Multiplayer Game
It is interesting to see that Counter-Strike is the most popular multiplayer gameplayed online, with anywhere from 25,000 to 60,000 simultaneous players. One could say that Half-Life itself was a mega-hit with over two million copies sold, whereas Rainbow Six was a more modest success, and use that argument to explain why Counter-Strike is the more popular counterterrorist game. However, that argument fails when you realize people do not play games they do not want to play. Sure, marketing can help a game get off the ground to some extent, but the games business is still dominated by word-of-mouth sales where one fan recommends the title to another. The big titles that receive large marketing budgets are also fun and playable games that enjoy strong word-of-mouth sales. Unlike the movie business, an aggressive marketing campaign cannot save your bacon.

There is a long-standing tradition of going to bad movies just to see how bad they are; this does not happen with games. Games are too expensive at about $50; no one is inclined to buy a game just to see how bad it is. However, a bad movie has a couple of chances. First of all, just seeing what mischief with toddlers Arnold Schwarzenegger has gotten himself into complete with some buttered popcorn, a fountain soda, your friend's company, and a walk about the mall is a good entertainment value. This movie will go onto DVD, VHS, rental, cable, then prime-time TV, and eventually the USA channel-plenty of ways for a non-hit movie to recoup and make a small amount of money for the studio.

The 50,000 people playing Counter-Strike online is even more impressive when you think about the ratio of people playing the multiplayer portion of a game relative to the single-player portion. It has been casually measured across a number of games, excluding the massively multiplayer online role-playing games, that only about 5 to 15 percent of the purchasers of a game will go on to play it in its multiplayer format. Thus Counter-Strike was much more successful than Rainbow Six, and it was working with only 5 to 15 percent of the counterterrorist market.

Of Intersecting Sets and Elite Forces

A second-tier game will sell its most copies in the first few weeks when the early adopters who have kept on top of all the previews will buy the game. During this time period the online reviews are written up. To my surprise it appears that strong reviews cannot sell a game either. The most excellent Elite Force (not anywhere close to being a second-tier game) developed by Raven received the most stellar press reviews one could ask for, including game of the year from most publications. Built on the Quake engine and developed by a top developer, it had lavish press coverage generating plenty of awareness before the release of the title. The title was reasonably on time and reasonably bug-free. The team behind the game was so into the game, they produced a free expansion pack. Elite Force was firmly expected to be a major hit inside of Activision. I do not know the actual numbers on the internal return-on-investment worksheets, but I have heard they were expecting 700,000 to 1,000,000 units in the first year worldwide. Elite Force went on to do about one-third of those numbers. Why? Why did Elite Force not succeed when not a single person at Raven, Activision, or the press could have set the game up better for success? Is it bad luck? Is the gaming public so fickle?

I have a theory why Elite Force failed to meet Activision's expectations. First of all, the game did sell well at approximately 300,000 units generating a gross revenue of $15 million. That is enough money to make a living for all involved and keep at it. However, I think it is the expectations that were at fault; I don't think the game could ever hope to sell more units than it did. Sure a truly immense advertising campaign with television commercials played 20 times a day on all channels and appearances of the game on all of the late-night talk shows would have sold maybe 100,000 to 200,000 more copies, but Activision would have had to pay for each copy they were selling. My theory is that when you are experimenting with genre crossing and
blending, be sure you are creating a union between the two or more sets of players you are marketing to, and not creating the intersection between these markets.

RAY MUZYKA SPEAKS: This certainly is an art form, but I think it can be done; it's just difficult. Creating the correct impression on the fans of both genres and making the parts that don't appeal to the other genre's fans at all times accessible is probably the hardest thing to implement, but this is critical to achieving mainstream success through selling to a few hard-core genres in a cross-genre game.

The two markets for Elite Force were the Star Trek gamers and the first-person shooter gamers. Activision has been working hard for years trying to find a breakaway hit for the Star Trek license they paid so dearly for, and teaming up with world class developer Raven and using the fabulous Quake engine should produce a lavish 3D-game with production values far and above any that a Star Trek gamer has seen before. And for the first-person shooters who are tired of blowing monsters up in worlds freshly created with little or no backstory, Elite Force offered the Star Trek universe, which consumers have had exposure to for over 25 years. Sounds wonderful, so why did this game not sell a million copies or more? Warcraft II was just a sequel to a game of orcs and humans gathering rocks and trees and banging on each other. That sold millions of copies; why shouldn't Elite Force sell a million? The reason is in the key design elements themselves; the very strategy used to make a hit—a cross between Star Trek and first-person shooters—is what held Elite Force back.

Let us first take a look at Elite Force from the perspective of a Star Trek gamer. Star Trek is about a starship named Enterprise exploring the galaxy on romantic adventures that are solved through cleverness, diplomacy, or the gunboat diplomacy that the Enterprise can deliver with photons and phasers. The Star Trek gamer is looking to live the experience depicted in the television episodes and movies. These episodes feature fantastic science, starship combat, and exploring various social themes in a futuristic context. Star Trek does feature combat between individuals in the form of the hand-held phaser, a device that you just point and shoot to disable or to disintegrate. This weapon reveals an utter disdain for prowess of personal martial skill; this hand phaser is almost a nerd fantasy where they can get back at every childhood bully by just pointing their garage door opener-and bzzt!-no more enemies. The Star Trek gamer is not looking for a first-person shooter; there is nothing in the Star Trek universe backstory that leaves the player wanting to explore a shooter. The most successful Star Trek games have been the adventure games 25th Anniversary and Judgment Rites, as well as the starship games of Starfleet Command, Starfleet Academy, and Armada.

From the first-person shooter perspective, an FPS player traditionally looked for the technically impressive and challenging games such as the Quake and Unreal series. However, after the release of the story-rich Half-Life, the industry realized that the FPS crowd would love to have a good reason to exercise their martial prowess. The creepy world of Half-Life is a good reason, the pulse-pounding excitement of World War II through Day of Defeat is a great reason, and hunting terrorists with a submachine is always great fun. But again the Star Trek universe lacks any compelling imagery of personal combat. Sure, Kirk would slug it out with the occasional alien, and Spock could put someone to sleep by pinching them; either way, Star Trek lacks that visceral appeal.

Star Wars, on the other hand, has a glorious tradition of martial combat on the personal scale through the use of light sabers. This style of combat was indeed a strong success with the Jedi
Knight series from LucasArts. Finally, let me repeat, Elite Force was not an unsuccessful game; it was a great game, very well produced. And missing the expectations set for it is not a reflection on the execution of Elite Force, but rather a reflection on the key design concepts of the game.

Some Straight Questions to Ask Yourself

The case studies I presented introduced use cases from the Unified Modeling Language and illustrated what I mean by determining the key design elements of your game. I ask you to pause just a moment before you wield your scalpel and slice off the most extraneous bits of your game design. I would like you to first get a bit more material down on a second sheet of paper to consider while you review your key design elements.

- **What Genre or Genres Does Your Game Feature?**
  First, what is your game's genre, such as adventure, role-playing game (RPG), real-time strategy (RTS), real-time tactical (RTT), action, first-person shooter (FPS), puzzle, sports, or some other genre? Or is it a blend of genres? Write down your game's genre or genre blend, and why.

- **Will the Game Be Single-Player, Multiplayer, or Both?**
  Does your game play well as a single-player game but perhaps not make much sense as a multiplayer game? Or is it the other way around where it takes real humans to play against to make it fun? Or is it reasonably fun either way? Write down single-player, multiplayer, or both, and why.

- **What Is the Platform?**
  Which platform are you targeting: PC, handheld, Xbox, PlayStation 2, or GameCube? Write down the platform or platforms you are targeting, and why.

- **What Is Your Target Market?**
  Is this a game anyone could enjoy? Or is it targeted for the core game market of males 18 to 45 years of age? Are you targeting women as well as men? Children? What is the violence level in your game? The language? Sexual content? Write down your target market, and why.

- **What Major Technologies Are You Using?**
  Is your game to be 2D or 3D in its fundamental presentation? Will it use a commercial engine? Is there something special about the physics? Perhaps you envision cell-shaded rendering of characters or the scene. Write down the major bits of technology you will employ in your game, and why.

Now What?

Notice I did not give any opinions or suggestions on how to answer those questions or which answers I thought you might choose. It is not my place to tell you that a cell-shaded 3D RPG would be the next big thing on the Game Boy Advance. No, the answer to the questions above need to come from your heart, that place of inner vision where you can see and play your game in your mind's eye. That gameplay in your mind—I want you to write that down. This is your game. If you told me your game concept, I could offer suggestions and opinions, but they would be just that—opinions and suggestions. For this game of yours to be a success you must be able to have a strong vision for how your game will play.
Now find a table someplace comfortable and put in front of you the notes you have taken on game concept, business context, and the feature questions asked above. Then I want you to put this book aside and just keep visualizing your game. Get up and take a walk, get something to eat, and come back to your table of notes. Now, start slicing out the parts of your game feature brainstorm that are not actually central to your game design. Before you invest in creating a hundred-page game design document and develop a total technical design, you should figure out what you are making. The game design and technical design stages are a lot of work; be courageous and kill the features that are superfluous before you spend any more effort on them.

All of the great games have a small feature set that is well polished. Make your game great.

Copyright © 2003 CMP Media Inc. All rights reserved.