The Case For Game Design Patterns

By Bernd Kreimeier
Gamasutra
March 13, 2002

URL: http://www.gamasutra.com/features/20020313/kreimeier_01.htm

Game design, like any other profession, requires a formal means to document, discuss, and plan. Over the past decades, the designer community could refer to a steadily growing body of past computer games for ideas and inspiration. Knowledge was also extracted from the analysis of board games and other classical games, and from the rigorous formal analysis found in mathematical game theory.

However, while knowledge about computer games has grown rapidly, little progress has made to document our individual experiences and knowledge - documentation that is mandatory if the game design profession is to advance. Game design needs a shared vocabulary to name the objects and structures we are creating and shaping, and a set of rules to express how these building blocks fit together.

This article proposes to adopt a pattern formalism for game design, based on the work of Christopher Alexander. Alexandrian patterns are simple collections of reusable solutions to solve recurring problems. Doug Church's "Formal Abstract Design Tools" [11] or Hal Barwood's "400 Design Rules" [6,7,18] have the same objective: to establish a formal means of describing, sharing and expanding knowledge about game design.

From the very first interactive computer games, game designers have worked around this deficiency by relying on techniques and tools borrowed from other, older media -- predominantly tools for describing narrative media like cinematography, scriptwriting and storytelling. Computer games are a visual medium, and game designers are increasingly relying on design techniques developed for movies - to the detriment of efforts to identify methods genuinely suited for game design. The discussion of narrative techniques has come to dominate the discourse on game design, almost extinguishing alternatives.

If the techniques borrowed from other media were sufficient to express and address game design issues, there would be no need to search for alternatives. However, the metaphors and devices

borrowed from narrative media are usually insufficient (or even inadequate) to capture the essence of the interactive game medium. The community is aware of this [12], although the response is often an attempt to reconcile the contradiction by redefining the term "narrative" [24]. Taken to the extreme of an artificial playwright [23], this constitutes proposing a technological solution to a conceptual problem. The real issue is not the shortcomings of narrative techniques with respect to their utility in game design, but the lack of techniques genuinely suited for interactive media.

The game design pattern method proposed here is concerned with content patterns, as opposed to software engineering patterns [19], specializations of which that have been proposed for game programming [33,20]. Similarly, process patterns to organize and manage game development projects (such patterns could be extracted from [17,28]) are beyond the scope of this article.

What Are Patterns?

In a nutshell, patterns are simply conventions for describing and documenting recurring design decisions within a given context, be it game design or software engineering. Specific patterns are the result of applying this method consistently, leading to collections of design patterns which have been assigned a name and are documented by an anecdotal or abstract description. Pattern methods provide semi-formal tools for problem domains in which rigorously formal methods cannot easily be applied, or are simply not available or even conceivable.

Patterns are traditionally expressions of problem-oriented thinking. The seminal book by Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, quotes Alexander [3]: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of a solution to that problem, in such a way that can you can use this solution a million times over, without ever doing it the same way twice." There can be several alternative solutions to a given problem, each one defining its own pattern, but the combination of problem statement and solution proposal is the essence of any Alexandrian pattern.

A game design pattern collection would provide a shared design vocabulary that allows experienced designers to:

- communicate efficiently with each other, with less experienced designers, and with members of other professions (like software engineers and game coders)
- document their insights, organizing individual experience as written knowledge
- analyze their own design as well as the designs of others, e.g. for purposes of comparative criticism, re-engineering, or maintenance

It is important to distinguish between pattern-based methods, which are very generic and general, and specific pattern collections created for a given purpose. The decision to use patterns merely determines form, not content. The conventions of any pattern template do not guarantee (or prohibit) that useful patterns will be found and documented. Pattern methods are simply a successful way to express existing knowledge.

A Pattern Template

Pattern templates typically contain these four essential elements:

1. **Name.** "Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction". Names have to be mnemonic and evocative, but the connotations also pose problems. "Also Known As", frequently part of pattern templates, is actually an indication of a naming problem: "Finding good names has been one of the hardest parts of

developing our catalog" (again, Erich Gamma et al.).

2. **Problem.** This describes the problem, including its inherent trade-offs and the context in which the problem occurs. The description of the problem implies a goal that we want to accomplish, and the obstacles we encounter when we attempt to do so.
3. **Solution.** A description of a general arrangement of entities and mechanisms that can be used to solve the problem. This is not a particular design or concrete implementation, but an abstract structure that describes an entire family of solutions that are essentially the same.
4. **Consequences.** Each solution has its own trade offs and consequences. Solutions can, in turn, cause or amplify other problems. The costs and benefits of a solution should be understood and compared against those of alternatives before making a design decision. Around this essential core, pattern templates often add other elements, or subdivide a core element.

Gamma et.al. uses Name - Intent - Aliases - Motivation - Applicability - Structure - Participants - Collaborations - Consequences - Implementation - Sample - Known Uses - Related Patterns . Meszaros and Doble describe pattern writing itself in patterns organized as Title - Problem - Context - Forces - Solution - Indications - Resulting Context - Related Patterns - Examples - Samples - Rationale - Aliases - Acknowledgements [25]. Alexander et. al. use Name - Example as Picture - Context within larger patterns - Problem - Solution - Solution as Diagram - Relation to smaller patterns [3]. Ultimately, the details of the format chosen do not matter as much as the fact that one format has been elected and is used consistently (for more details on how to write patterns, see [31,34,5]).

Pattern Examples

To show the method in action, let's look at some pattern candidates. The following examples are admittedly very simple, and weren't selected necessarily for their particular relevance to game design.

PROXY

Problem: It might not be possible or desirable to require that a game action is applied to the target object directly. In other words, we have to avoid proximity to or collision with the ultimate target of a given action.

Solution: Introduce another entity (object) that is used as a proxy, standing in for the target object. The proxy can be placed and moved independently of the target object or objects. The restrictions that apply to the target(s) do not by default apply to the proxy. Multiple proxies can be used for the same target or targets.

Consequence: The link between the proxy and the actual target has to be communicated to the player, if the player is supposed to exploit this connection and indirection. Restrictions on placement of proxy and target with respect to player field of view, and restrictions on player view control might apply, to ensure that the player perceives the effect on the target.

Examples: *Munch's Oddyssee* has several proxies for the player characters: the possession orb which enables Abe to take over other game characters (thus turning them into proxies), and the Snoozer robot and H-Crane, which served as proxies for Munch. One benefit of player character proxies is that of a safe death: loosing a proxy does not end the game. Buttons/Switches/Levers that open doors or start/stop machinery are often placed in different locations, instead of being part of the target object. In

these cases, the designer wants to separate trigger from target to permit more elaborate puzzles. Half-Life's final boss had a vulnerable location inside its head. That vulnerable spot could be considered a proxy for the entire (invulnerable) boss. As a second order vulnerability, the boss was dependent on three crystals that supplied energy for healing and attack. Attacking the crystals destroyed them, thus permitting effective attacks on the boss monster itself.

PROXY was chosen to illustrate potential problems in using the same name for different patterns existing in different contexts. There is a software engineering pattern with the same name, introduced by Gamma et al. However, there is no meaningful connection between the two concepts. On the other hand, in some cases a given pattern language might translate into a sensible OOP implementation hierarchy, if there is a correspondence between the purpose of a given game design entity within the design context, and its implementation in the software that constitutes the game engine.

PREDICTABLE CONSEQUENCE

Problem: The player has to perceive failure as a consequence of her mistakes, not as a random or predestined event unrelated to, and unavoidable by, any reasonable choice of actions on her part.

Solution: The player cannot take a meaningful decision to act (or not to act) if the result of a possible action can not be anticipated. A meaningful player decision is an informed decision: she has to be able to guess the result of her action before she ever takes it. Choose game mechanisms that communicate predictable behavior by visual appearance relying on knowledge the player already has (either from real world experience, or from experience with other games, or from experience with preceding parts of your game). Do not break consistency of your visual language, i.e. make sure that objects that behave identical look identical.

Consequence: It is no longer possible to implement surprises that are impossible to anticipate. The player might rely on the out-of-context assurance to anticipate the designer's ambush, and either outwit the designer, or find herself locked into performing an action against her better judgment as the designer has prevented alternative courses of action. Ultimately, the designer might arrange for actions that the informed player will never perform, as they are predictably to her disadvantage. The designer might have to go to considerable length to communicate the rules that govern the outcome of actions, to the player. The design might be biased towards using mechanisms resembling real world objects, to minimize the need for explanations. A lack of uncertainty on the player's side makes the game more transparent, potentially removing the need for exploration and experimentation. Ultimately, the need for Predictable Consequence is determined by the need

for backtracking (severity of failure) and costs of backtracking (latency between choice and consequence, see also behavioral science on timing issues). Another possible consequence is that the design might have to restrict itself to a smaller set of player actions, in order to be able to enable them consistently throughout the game world. Actions only available once or in a very few locations do not amortize the costs of communication and teaching/learning on the designer's/player's part. The design might also have to exaggerate the differences between distinct objects to avoid ambiguities. Any kind of variation or range has to follow rules that can easily be observed and complied with.

Examples: Doug Church's article relied on *Mario 64* as the textbook example [11] of predictable consequence. This pattern could be applied to any game with a consistent physics approximation, such those exhibited by bouncing grenades in FPS games. Other examples include crates and barrels that are marked as explosive actually explode, fireballs that hurt a player when they hit, burning fire, water that drowns if you do not swim, and lava which kills. Counter-examples are plentiful: switches and buttons that say "Push Me" to unleash a necessary ambush (an ambush that also opens the only way to move on) anticipated only as such. Openings in which to jump without knowing what is on the other side such as those found in *Opposing Force* and *Ico*. This is often a necessity imposed by the level partition/transition mechanism the engine uses.

References: Doug Church introduced the concept of perceived/perceivable consequence [11], which he uses also do describe predictable outcomes. There is a difference between the requirement for immediate feedback that the player can connect to her actions, and the requirement that the player has to have a chance to anticipate the likely outcome of her action. This pattern was derived from the latter, the former would require another pattern.

PREDICTABLE CONSEQUENCE is my rewrite of FADT "Perceivable Consequence" introduced by Doug Church (see [11] for the original description). Aside from changes to the description itself to refine its meaning (as I perceived it), this pattern candidate illustrates that FADTs can expressed as patterns.

PAPER-ROCK-SCISSORS

Problem: Avoid a dominant strategy that makes player decisions a trivial choice.

Solution: Introduce nontransitive relationships within a set of alternatives, as in the game of paper-rock-scissors.

Consequence: The player is no longer able to find a single strategy that will be optimal in all situations and under all circumstances. She has to revisit her decisions, and, depending on the constraints imposed by the game, adjust to changing situations, or suffer the consequences of an earlier decision.

Examples: The example given by Andrew Rollings is the set of warrior-barbarian-archer from the Dave and Barry Murray game *The Ancient Art of War* (Broderbund 1984). He also describes *Quake*'s weapon/monster relations in similar terms: Nailgun beats shambler, shambler beats rocket launcher, rocket launcher beats zombie, zombie beats nailgun [28].

References: Chris Crawford (see "Triangularity" in [15]) provided the first explicit description of the use of nontransitive relationships. Andrew Rollings' discussion of examples uses game theory including detailed payoff, as well as informal fictional designer dialogs.

PAPER-ROCK-SCISSORS is a recurring game design device that may have been the very first to be semi-formally described. It can also be presented as a pattern. There is no new information in this rewrite; the existing knowledge has merely been reorganized to match the other examples. The pattern could be extended to include a description of payoff matrices and examples for sets larger than three (see [28] for such a discussion), but even the minimal description captures its essence.

PRIVILEGED MOVE

Problem: Sometimes a given game entity cannot be permitted to interfere with others, or other entities cannot be permitted to interfere with it.

Solution: Introducing exclusive moves and locations separates and protects game entities. These exclusive moves and locations cannot be entered or left without being able to perform the respective privileged move. By introducing different, separate spaces or media, the game environment is broken down into distinct areas, which can serve e.g. as safe zones or safe passages.

Consequence: Depending on the implementation, this can be a very heavy-handed way to ensure protection. Unless protection is implemented in a way that is at the player's disposal (a door that only she can open and close), the player might perceive the constraint as annoyance and disappointment. If the restriction is meant to be temporary, the mechanism to allow it to be lifted has to be communicated to the player, and initiated by her.

Examples: In *DOOM*, monsters could traverse acid pools, but the player could only at the expense of damage that quickly turned to be lethal. In *Half-Life* and *Halo*, enemy soldiers can be dropped from flying vehicles, a transportation mechanism not available to the player, and impossible to interfere with. *Half-Life* also had a spider-like boss that was capable of breaking through its own nets, which blocked the player's way until she forced the monster to retreat. In *Thief*, the player traverses rooftops in relative safety, permitting preview and outlook on the level bottom. In *Ico*, the ghostly opponents traverse the world in a different dimension, and emerge from portals without apparent restriction. In *Munch's Oddyssee*, water was inaccessible to enemies, providing one player character a privileged swim move and safe passage (in the absence of snipers and mines). In many games, water enemies are in turn restricted to that medium (e.g. *Quake*, *Half-Life*). In many games flying opponents (e.g. the harpies in *Heretic 2*) can perform moves the player cannot, and thus enjoy some amount of safety.

PRIVILEGED MOVE is a pattern proposal for a mechanism rarely discussed, but frequently used for a multitude of purposes. A single solution can solve or address multiple problems; a situation that is not clearly addressed in Alexandrian pattern languages that emphasize the problem-oriented aspect of pattern use. PRIVILEGED MOVE is also a very close relative of FILTER. This example set is by no means comprehensive. As Gamma et al. wistfully remark: "Finding patterns is much easier than describing them."

FILTER

Problem: If the player is given means to create new game objects, and/or new combination of existing objects, and/or move game objects arbitrarily, the resulting complexity might become overwhelming for the designer.

Solution: Limit local complexity by filtering, thus creating smaller, clearly defined subsets that can be dealt with.

Consequence: Player freedom is restricted by implicit or explicit means. Accidental filtering can lead to game situations that the player cannot solve. Ideally, the filtering is obvious to the player, and inherent part of the game world. FILTER can be an intended consequence or unintended side effect of PRIVILEGED MOVE.

Examples: FILTER as a means of dealing with combinatorial complexity of possible game states is extracted from Jon Blossom's postmortem of *DroidWorks* ([9]). Quote: "Our level designers [..] could never be entirely sure what kind of droid would be walking into their rooms [...] In many cases, they created ingenious physical filtering mechanisms that would guarantee only certain types of droids went beyond certain points in the level: A steep hill would weed out biped droids in favor of droids with tractor treads, a chasm would weed out wheeled droids that couldn't jump, a narrow canyon would weed out wide droids, and a short door would weed out tall droids. The designers used the terrain leading up to key puzzles to reduce the complexity of the puzzle itself [...]"

WEENIE

Problem: Players might loose their sense of direction with respect to how the game world unfolds. This is a particular problem with player-driven scripting proceeding in lockstep with player actions.

Solution: The game has to establish clear leads that communicate to the player where to go, and what actions to attempt once there.

Consequence: The player will over time come to depend on the level of guidance, and will be confused if this guidance is dropped or omitted, which introduces a bias towards WEENIE CHAIN. Depending on the rigidity of the guidance mechanisms, players might become aware the out-of-game agenda behind the in-game setups.

Examples: On occasion this has been called the "carrot on a stick" approach to level design [8]. Stephen Clarke-Wilson [13] explains: "This somewhat bizarre term was coined by Walt Disney, who suggested that when designing massive 3D environments (theme parks), it was necessary to lead visitors through the environment the same way one trains a dog-by holding a wiener and leading the dog by the nose. Obvious weenies at Disneyland are Sleeping Beauty's Castle, which encourages guests to travel from the main entrance to the central hub; the former Rocket Jets, which encourage guests to explore Tomorrowland; the Mark Twain Steamship and dock, which encourage guests to explore Frontierland; and the King Arthur Carousel, which encourages guests to walk over the castle moat and into Fantasyland. [...] Your 3D VR environment needs to have standout landmarks so that it's easy to navigate without a map. The best games, which have typically been designed with very limited graphics, always save a few graphics to denote special and interesting things that should be investigated." Alternative means to communicate to the player the next proposed or mandatory step are explicit messages delivered by NPCs, or brief camera cuts that draw attention to doors opening in nearby locations, like that used in *Munch's Oddyssee*. Cliff Bleszinksi describes as the most subtle and always present WEENIE the incentive of seeing a new area: the player is always inclined to move from "been here, done this" into unknown territory [8].

WEENIE CHAIN

Problem: The player can lose sense of direction in the absence of clear indicators of where to go next. This is particularly a problem in games with large or non-linear environments that do not rely on rails or cut-offs to constrain player movement, or employ "revisiting" of areas [8].

Solution: Create an uninterrupted chain of WEENIEs, with each link of the chain clearly perceivable from its predecessor, guiding the player from start to finish.

Consequence: WEENIE CHAIN might be limited in employing branches or otherwise introduce ambiguity or player choice. Enforced, WEENIE CHAIN might resemble a rail.

Examples: Noah Falstein refers to *Indiana Jones and the Fate of Atlantis* as an example of how to establish a chain of clear player goals using messenger NPCs [18]. His other examples are *Super Mario 64* and its use of signs and NPCs, as well as Halo. *Munch's Oddyssee* also uses signs, as well as a Shaman NPC for initial guidance.

Pattern candidates can be harvested from many sources. For example, the pattern FILTER is taken verbatim from a game postmortem. WEENIE is taken from a discussion of theme parks and techniques of environmental storytelling, and has been presented under a different name in lectures on level design. WEENIE also is the foundation of WEENIE CHAIN, illustrating pattern dependency, composition, and hierarchy.

Once you start looking for patterns, you start noticing them everywhere. Erich Gamma et al. point out that whether they are aware of it or not, novelists and playwrights already use patterns. Many "narrative devices" (see e.g. [32]) could also be described using a pattern template.

Sometimes the need to differentiate patterns on distinct levels of abstraction leads to different labels: "One person's pattern can be another person's primitive building block" (from [19]). Alexander et al. present a total of 253 patterns, subdivided into three sets for "Towns, Buildings and Construction" [3]. Preferences regarding the virtues of a given purpose might apply: pattern are just recipes, means not ends, and disagreement on the ends is beyond the scope of a pattern. Designers might also have different views on the quality of a given solution: one person's pattern might well be another's "anti-pattern" -- a recurring example of a bad design decision. Ideally, the pattern description itself is just a candid summary of cause and effect, describing one way to reach a given objective.

Alexandrian patterns are not unknown to game developers. Many programmers are familiar with Gamma's concept of software design patterns. Will Wright of Maxis credits the same book as the original inspiration for The Sims, and refers to Christopher Alexander's works (namely [1]). Steven Chen and Duncan Brown, former level designers at LucasArts, refer level designers to the 253 Alexandrian patterns merely as a guideline to create "meaningful environments" [10].

Instead, game developers strive to develop their own forms and templates. The Formal Abstract Design Tools (FADTs) proposed by Doug Church can easily be rewritten to fit the canonical pattern template. Another recent attempt to document game design knowledge in semi-formal ways is "The

400 Project" by Hal Barwood and Noah Falstein, is a project to collect proven game design rules and techniques. This 400 Project dates back to a GDC 2001 lecture by Hal Barwood, in which he also presented four examples.

Their effort has now spawned a column in *Game Developer* magazine, in which Noah Falstein introduced the rule "Provide clear short-term goals" as the opening example. The 400 Project uses a five-part template: Imperative Statement - Domain of Application - Dominated Rules - Dominating Rules - Examples Aliases (see [18] for details). The concept of dominance (that some rules take precedence over others, and might in turn be trumped themselves) is absent from Alexandrian patterns. An Alexandrian pattern simply lists one possible solution to a given problem, acknowledging that other solutions might exist. Alexander sees each pattern as a recipe that strives to balance competing imperatives and conflicting forces in order to achieve the best possible results with respect to a specific objective. The decision to use a given pattern will also depend on whether there are multiple objectives to fulfill or when aside effects of applying a given solution, influence the. Falstein's rules appear more rigid, implying that all games have the same goals, and the same concerns. The names of the rules in the 400 Project are imperative statements, like Provide clear short-term goals, and their description contains only the solution part of a pattern. The problem to be solved by the pattern is merely implied or presumed obvious in this revision of the rules. WEENIE, as well as WEENIE CHAIN, capture parts of Provide clear short-term goals but also explicitly state the role of the pattern. Like rules, pattern names typically name the solution, not the problem: a matching pattern might just be called SHORT-TERM GOALS.

Outlook

Patterns are a formal means of documentation, and such means open the door to software tools for maintaining and editing game design documents. Take screenplays: word processors can be configured to handle the canonical format for scriptwriting, and some applications have been designed specifically to support that movie industry format. Many game companies and game designers have devised their own internal standards for game design documents. But defining a standard format for game design documents is of limited use unless there are editing and search facilities that support and enforce the format.

A pattern language is a natural match for descriptive markup, e.g. XML, with a huge potential for tool support based on off-the-shelf software. The value of any "living" document is directly related to the ease of its maintenance. Structured editing aids structured thinking: if a design document does not have clear organization, its use of keywords and names is inconsistent, and relevant information can not be located quickly when needed, the document is practically useless.

Consequently, game developers have to make a sustained, conscious effort to define and describe the recurring elements of their daily work - whether as patterns, rules or some other method -- so we can begin to create software tools made or adapted specifically for game design purposes. The case for Alexandrian game design patterns [22,29] seems strong: they have proven themselves in other and diverse professions; they are intuitive, well documented, and a familiar concept to software engineers, yet are flexible enough to permit anecdotak or informal descriptions of artistic choices.

Bibliography

[1] Christopher Alexander. *Notes on the Synthesis of Form*. (Harvard University Press 1964, 1966, 1979.) ISBN 0-674-62750-4 (cloth) ISBN 0-674-62751-2 (paper)

[2] Christopher Alexander, Murray Silverstein, Shlomo Angel, Sara Ishikawa, and Denny Abrams. *The Oregon Experiment*. (Oxford University Press, 1975.) ISBN 0-19-501824-9

[3] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction*. (Oxford University Press, 1977.) ISBN 0-19-501919-9

[4] Christopher Alexander. *The Timeless Way of Building.* (Oxford University Press, 1979.) ISBN 0-19-502402-8

[5] Brad Appleton. *Patterns and Software: Essential Concepts and Terminology*. Last update: Feb. 2000.

[6] Hal Barwood. "Four of the Four Hundred 2001". (GDC lecture, 2001.)

[7] Hal Barwood and Noah Falstein. "More of the 400: Discovering Design Rules 2002" (GDC 2002 lecture)

[8] Cliff Bleszinski. "The Art and Science of Level Design." (GDC 2000, pp. 107--118.)

[9] Jon Blossom and Collette Michaud. "Postmortem: LucasLearning's Star Wars DroidWorks" (Gamasutra 1999.) Originally *Game Developer* magazine, Vol 3, Issue 28, pp. 52-58, July 1999.

[10] Steven Chen and Duncan Brown. "The Architecture of Level Design." (GDC 2001 Proceedings, pp. 167--175.)

[11] Doug Church. "Formal Abstract Design Tools." (Gamasutra, 1999. Originally *Game Developer* magazine, Vol 3, Issue 28, July 1999.)

[12] Doug Church. "Abdicating Authorship: Goals and Process of Interactive Design." (GDC 2000, San Jose, Lecture 5403 (not in proceedings).)

[13] Stephen Clarke-Willson. "Applying Game Design to Virtual Environments" (*Digital Illusion*, ACM Press, Vol. 2, Issue 1, January 1, 1998.)

[14] (N/A).

[15] Chris Crawford. *The Art of Computer Game Design*, Chapter 6: "Design Techniques and Ideals." 1984.

[16] Troy Dunniway. "Using the Hero's Journey in Games." (Gamasutra, 1999. Originally published in *Game Developer* magazine, August 1999.)

[17] Troy Dunniway. *Professional Game Design.* (New Riders. To be published June 2002. ISBN 0-7357-1184-4.)

[18] Noah Falstein. "Better By Design: The 400 Project". *(Game Developer magazine*, Vol. 9, Issue 3, March 2002, p. 26.)

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. (Addison Wesley Longman, 1994.) ISBN 0-201-63361-2.

[20] Chris Hecker and Zachary Booth Simpson "Game Programming Patterns & Idioms." *Game Developer magazine*, Sep. 2000.

[21] John Hopson "Behavioral Game Design." Gamasutra, April 2001.

[22] Bernd Kreimeier. *Game Design Patterns*. Wordware Publishing, Inc. To be published March 2003.) ISBN 1-55622-967-4

[23] Brenda Laurel. *Computers as Theatre*. (Addison Wesley Longmasn, Inc. 1991, 1993 ISBN 0-201-55060-1.)

[24] Marc LeBlanc. "Formal Design Tools: Emergent Complexity, Emergent Narrative." GDC 2000, San Jose, Lecture 5304 (not in proceedings).

[25] Gerard Meszaros and Jim Doble "A Pattern Language for Pattern Writing"

[26] Pierre-Alain Mueller. *Instant UML*. (Wrox Press, Ltd., 1997) ISBN 1-861000-87-1.

[27] Karen Pryor. *Don't Shoot The Dog!* (Bantam Doubleday, 1999) ISBN: 0-55338-039-7 (revised paperback edition)

[28] Andrew Rollings and Dave Morris. *Game Architecture and Design*. (The Coriolis Group, 2000.) ISBN 1-57610-425-7

[29] Andrew Rollings and Ernest Adams. *Patterns in Game Design*. (The Coriolis Group, to be published May 2002.) ISBN 1-57610-873-2

[30] Richard Rouse. *Game Design: Theory & Practice*. (Wordware, Inc., 2000) ISBN 1-55622-735-3

[31] Aamod Sane "The Elements of Pattern Style." December, 1995.

[32] Viktor Shklosvsky. *Theory of Prose Dalkey*. (Archive Press 1990, 1991.) ISBN 0-916583-54-6 (cloth) ISBN 0-916583-54-6 (paper).

[33] Zachary Booth Simpson "Design Patterns for Computer Games." 1998 CGDC Austin, TX, November 1998, also San Jose, CA, May 1999.

[34] John Vlissides. "Pattern Hatching - Seven Habits of Successful Pattern Writers." *C++ Report.* Nov/Dec 1996, and *Pattern Hatching: Design Patterns Applied* (Addison Wesley, 1998).

[35] Christopher Vogler. *The Writer's Journey: Mythic Structure for Writers*. (Michael Wiese Productions, 1998.) ISBN 0-941188-70-1

The game design pattern method proposed here is concerned with content patterns, as opposed to software engineering patterns [19], specializations of which that have been proposed for game programming [33,20]. Similarly, process patterns to organize and manage game development projects (such patterns could be extracted from [17,28]) are beyond the scope of this article. What Are Patterns? In a nutshell, patterns are simply conventions for describing and documenting recurring design decisions within a given context, be it game design or software engineering. Specific patterns are the result of Design patterns have been a valuable asset to software developers for a long time. All kind of software, including games, benefit from re-using the well known solutions to the common problems. Building a game from scratch requires a carefully made design and this is where design patterns come handy. However, the approach to game development has changed lately.Â Other limitation is how well the choice of middleware to use for the case studies is justified. Cocos2d and Unity3D are two game engines referenced by this thesis, and they both are popular choices for making iPhone- and cross-platform mobile games, respectively. How well these engines represent the state-of-the-art tools for game development is up to the reader to find out. Augmented reality Game-design patterns Interaction patterns Learning games. This is a preview of subscription content, log in to check access. Notes. Acknowledgments. Parts of this work were supported by the European Commission under the Horizon 2020 Programme under grant agreement No 687669 (WEKIT). References. 1. Lamantia, J.: Inside Out: Interaction Design for Augmented RealityGoogle Scholar.Â Wetzel, R.: A Case for Design Patterns supporting the Development of Mobile Mixed Reality Games. Found. Digit. Games (2013). http://www.fdg2013.org/program/workshops/papers/DPG2013/b6-wetzel.pdf. Accessed 13 Nov 2017. 17.