Project Management Analysis Appendix

Game Project Management {#Game Project Management}

Everyone has an opinion on “how to create game-design documentation” and “how to manage game development”. Formal Project Management suggests the “Systems development life cycle” for software game design[^4] --writing “big designs up front” (BDUF) whose goal is to answer all questions about the game development process in a tome. The main problem with this formal process is the misconception of “perfect knowledge”. For small development studios such as ours, this formal process is expensive in time, man-hours, risk and money. In reality, one can never truly know everything about a game initially. In early stages of the game development process, one has the greatest range of “uncertainty”.

One gains more knowledge as development proceeds through the business rules and logic. The mistake of large development teams using formal project management is realized only too late in the postmortem follow-ups. The postmortem reveals:
1. Creating knowledge has a high cost in man-hours,
2. Knowledge is the greatest asset produced aside from the final game product itself. Is it a marvel that many small indie developers are moving to the new business model?

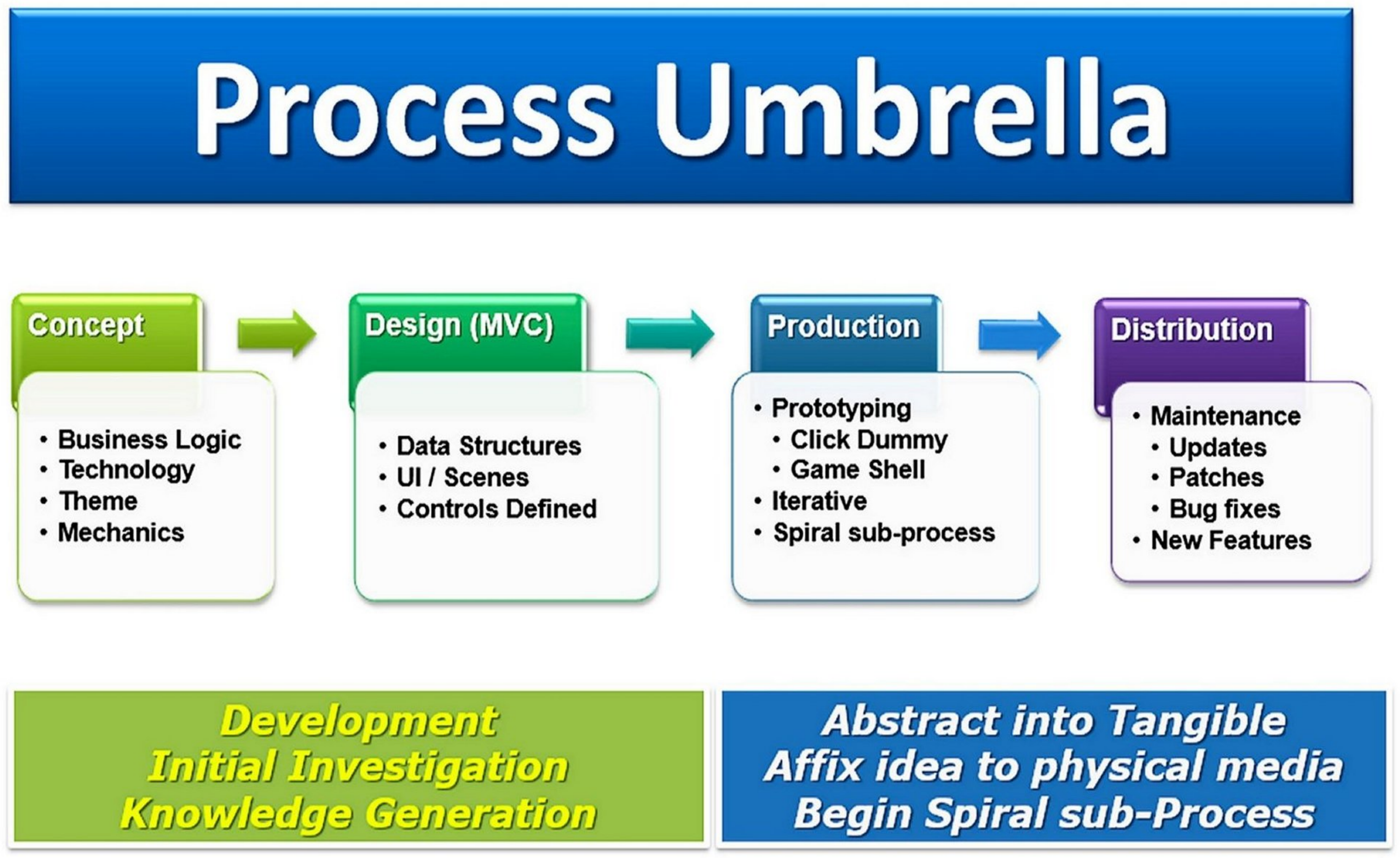
Flash is acclaimed as a rapid application development (RAD) environment. In general, the RAD approach to software development puts less emphasis on planning tasks and more emphasis on the development process. The RAD approach emphasizes the necessity of adjusting requirements in reaction to current knowledge gained as the project progresses. This relies on the use of throw-away prototypes in addition to or even sometimes in place of design documentation. If RAD is adopted, and under closer scrutiny, indie flash game developers simply use “Cowboy Coding” -- immediately producing source code. At some point, they would begin testing (often near the end of the production cycle), and the unavoidable bugs would then be fixed before distribution to “asset stores”. In essence, it is programming without a design; it could be labeled “design on-the-fly” or “wouldn’t that be cool, if ….” Too often “scope creeps” and excellent game ideas die in mid-development. Yet, indie developers have created revenues from simply a “game concept”.

Game Project Management (GPM) is becoming easier for smaller game indie to larger development teams. It seems the new trend is to integrate GPM into the game design editors.[^5] At SGGS, we use a GPM method known as “Software Prototyping” with “Extreme Programming”. On paper – also called “throw-away prototyping, we identify the game’s basic features and requirements in:

1. client and server technology,
2. business logic (i.e. Revenue generation),
3. environment themes, and
4. gaming mechanics.

We follow 4 core “umbrella” steps in our GPM:

- 1) Concept,
- 2) Design,
- 3) Production, and
- 4) Distribution.

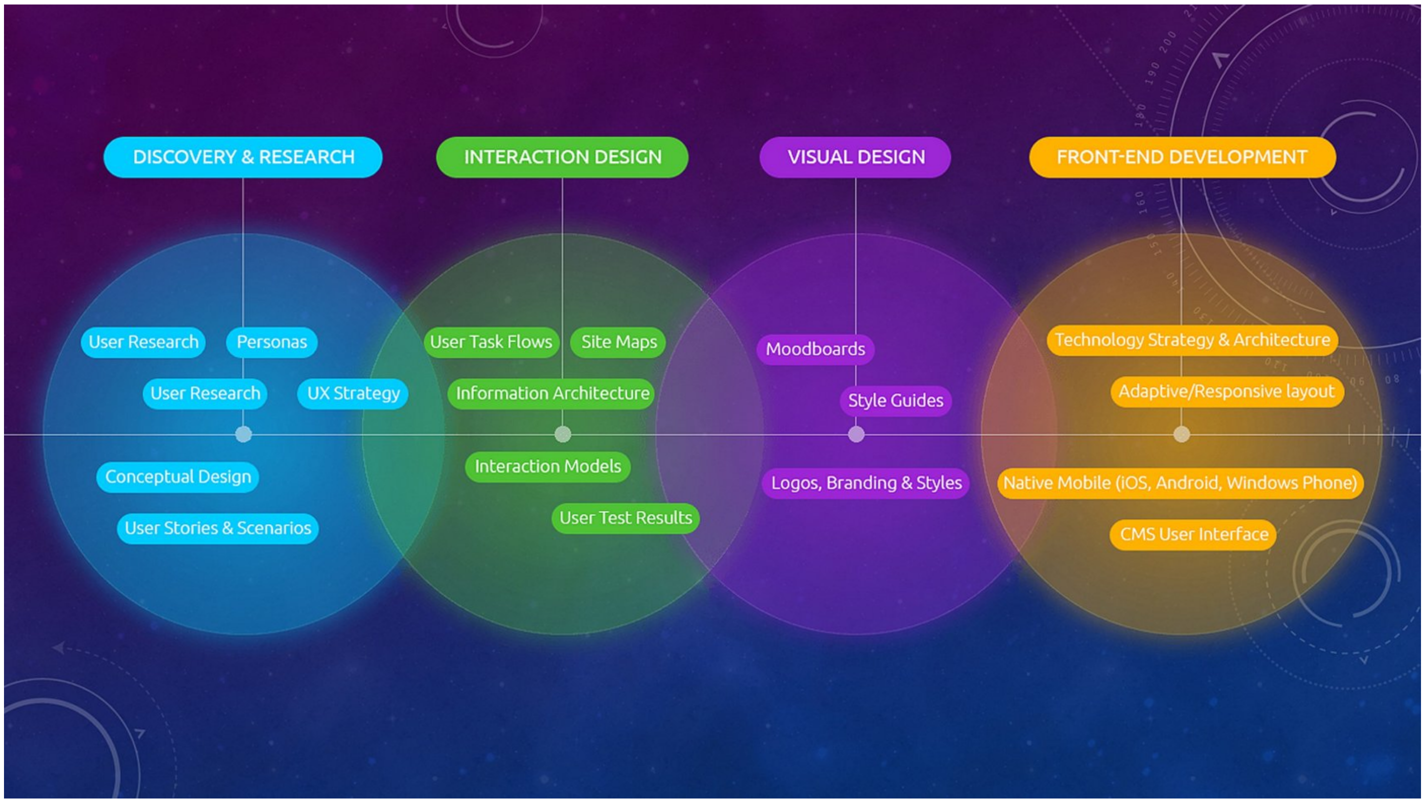


In the first two steps (we call "Development"), we are asking pre-production questions such as "what, how and is it fun". Some seasoned experts in the industry would say we are following a "Waterfall method" or possibly "SCRUM" project management – if labels are important to you. We are cutting the fat off SCRUM and using a "leaner more agile" model for our small studio development. Since our team consists of one, two, or sometimes as many as three people, we already have tight integration within multiple disciplines.
Capturing the ideas and research are the major activities in development. We generate knowledge before we enter the Production Phase. In this "development phase", we start with game logical data structures and build our data models ERDs. Next, we draft --on paper-- the game user interfaces (See User Interface Appendix), menus and navigation interactions. Afterwards, we move to create a "click dummy" of the game shell in the Flash CS IDE. It worth mentioning at this stage, we are moving from paper into physical design – labeling it "pre-production".

Now the fun starts, source coding! Iteration is paramount; development testing is usually done concurrently with, or at least in the same iteration as, programming. As we write procedures, frames, and functions, we write "just barely good enough" documentation so that we will understand what we're doing and the rationale why. Understand that we create games based on inspiration and that we may not return to a game idea for weeks, months or –in some cases—years. We have adopted a philosophy that game ideas alone won't help us and won't get us to the market. So, we create a working "game shell" component with fully operational navigation among scenes for ActionScript version 2 and/or 3. We spiral through each section of code until it works. We are using Flash as a "GUI Builder"[^6]. Creating this initial code, we often notice repetitive patterns or simpler ways to achieve the same end results by generalizing, pushing or pull content code; at this point, we are re-factoring.

Since 2010, we are considering the option to move our ActionScript source code off the main-timeline into external source files. This is a significant project with low priority. Your software source code product may appear either on the main-time or as external files depending on where that game is in its development stage.

Prototyping



Framework Type: Iterative

Basic Principles

1. Not a standalone, complete development methodology, but rather an approach to handling selected portions of a larger, more traditional development methodology (i.e., Incremental, Spiral, or Rapid Application Development (RAD)).
2. Attempts to reduce inherent project risk by breaking a project into smaller segments and providing more providing more ease-of-change during the development process.
3. User is involved throughout the process, which increases the likelihood of user acceptance of the final implementation.
4. Small-scale mock-ups of the system are developed following an iterative modification process until the prototype evolves to meet the users' requirements.
5. While most prototypes are developed with the expectation that they will be discarded, it is possible in some cases to evolve from prototypes toward a working system.
6. A basic understanding of the fundamental business problem is necessary to avoid solving the wrong problem.

Strengths:

1. "Addresses the inability of many users to specify their information needs, and the difficulty of systems analysts to understand the user's environment, by providing the user with a tentative system for experimental purposes at the earliest possible time." (Janson and Smith, 1985)
2. "Can be used to realistically model important aspects of a system

during each phase of the traditional life cycle." (Huffaker, 1986)

3. Improves both user participation in system development and communication among project stakeholders.
4. Especially useful for resolving unclear objectives; developing and validating user requirements; experimenting with or comparing various design solutions; or investigating both performance and the human-computer interface.
5. A potential exists for exploiting knowledge gained during earlier iteration as later iterations are developed.
6. Helps to easily identify confusing or difficult functions and missing functionality.
7. May generate specifications for a production application.
8. Encourages innovation and flexible designs.
9. Provides quick implementation of an incomplete, but functional, application.

Weaknesses:

1. Approval process and control is not strict.
2. Incomplete or inadequate problem analysis may occur whereby only the most obvious and superficial needs will be addressed, resulting in current inefficient practices being easily built into the new system.
3. Requirements may frequently change significantly.
4. Identification of non-functional elements is difficult to document.
5. Designers may prototype too quickly, without sufficient up-front user; needs analysis, resulting in an inflexible design with narrow focus that limits future system potential.
6. Designers may neglect documentation, resulting in insufficient justification for the final product and inadequate records for the future.
7. Can lead to poorly designed systems. Unskilled designers may substitute prototyping for sound design, which can lead to a "quick and dirty system" without global consideration of the integration of all other components. While initial software development is often built to be a "throwaway", attempting to retroactively produce a solid system design can sometimes be problematic.
8. Can lead to false expectations, where the customer mistakenly believes that the system is "finished" when in fact it is not; the system looks good and has adequate user interfaces, but is not truly functional.
9. Iterations add to project budgets and schedules, thus the added costs must be weighed against the potential benefits. Very small projects may not be able to justify the added time and money, while only the high-risk portions of very large, complex projects may gain benefit from prototyping.
10. The prototype may not have sufficient checks and balances incorporated.

Situations where *most* appropriate:

1. Project is for development of an online system requiring extensive user dialog, or for a less well-defined expert and decision support system.
2. Project is large with many users, interrelationships, and functions, where project risk relating to requirements definition needs to be reduced.
3. Project objectives are unclear.
4. Pressure exists for immediate implementation of something.
5. Functional requirements may change frequently and significantly.
6. The user is not fully knowledgeable.
7. Team members are experienced (particularly if the prototype is not a throw-away).
8. Team composition is stable.
9. The Project Manager is experienced.
10. No need exists to absolutely minimize resource consumption.
11. No strict requirement exists for approvals at designated milestones.
12. Analysts/users appreciate the business problems involved, before they begin the project.
13. Innovative, flexible designs that will accommodate future changes are not critical.

Situations where *least* appropriate:

1. Mainframe-based or transaction-oriented batch systems.
2. Web-enabled e-business system
3. Project team composition is unstable.
4. Future scalability of design is critic
5. Project objectives are very clear; project risk regarding requirements definition is low.

Incremental

Framework Type: Combination Linear and Iterative

Basic Principles:

> Various methods are acceptable for combining linear and iterative system development methodologies, with the primary objective of each being to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process:

1. A series of mini-waterfalls are performed, where all phases of the Waterfall development model are completed for a small part of the system, before proceeding to the next increment; OR
2. Overall requirements are defined before proceeding to evolutionary,

mini-Waterfall development of individual increments of the system, OR

3. The initial software concept, requirements analysis, and design of architecture and system core are defined using the Waterfall approach, followed by iterative Prototyping, which culminates in installation of the final prototype (i.e., working system).

Strengths:

1. Potential exists for exploiting knowledge gained in an early increment as later increments are developed.

2. Moderate control is maintained over the life of the project through the use of written documentation and the formal review and approval/signoff by the user and information technology management at designated major milestones.

3. Stakeholders can be given concrete evidence of project status throughout the life cycle.

4. Helps to mitigate integration and architectural risks earlier

5. Allows delivery of a series of implementations that are gradually more complete and can go into production more quickly as incremental releases.

6. Gradual implementation provides the ability to monitor the effect of incremental changes, isolate issues and make adjustments before the organization is negatively impacted.

Weaknesses:

1. When utilizing a series of mini-Waterfall for a small part of the system before moving on to the next increment, there is usually a lack of overall consideration of the business problem and technical requirements for the overall system.

2. Since some modules will be completed much earlier than others, well-defined interfaces are required.

3. Difficult problems tend to be pushed to the future to demonstrate early success to management.

Situations where *most* appropriate:

1. Large projects where requirements are not well understood or are changing due to external changes, changing expectations, budget changes or rapidly changing technology.

2. Web Information Systems (WIS) and event-driven systems.

3. Leading-edge applications.

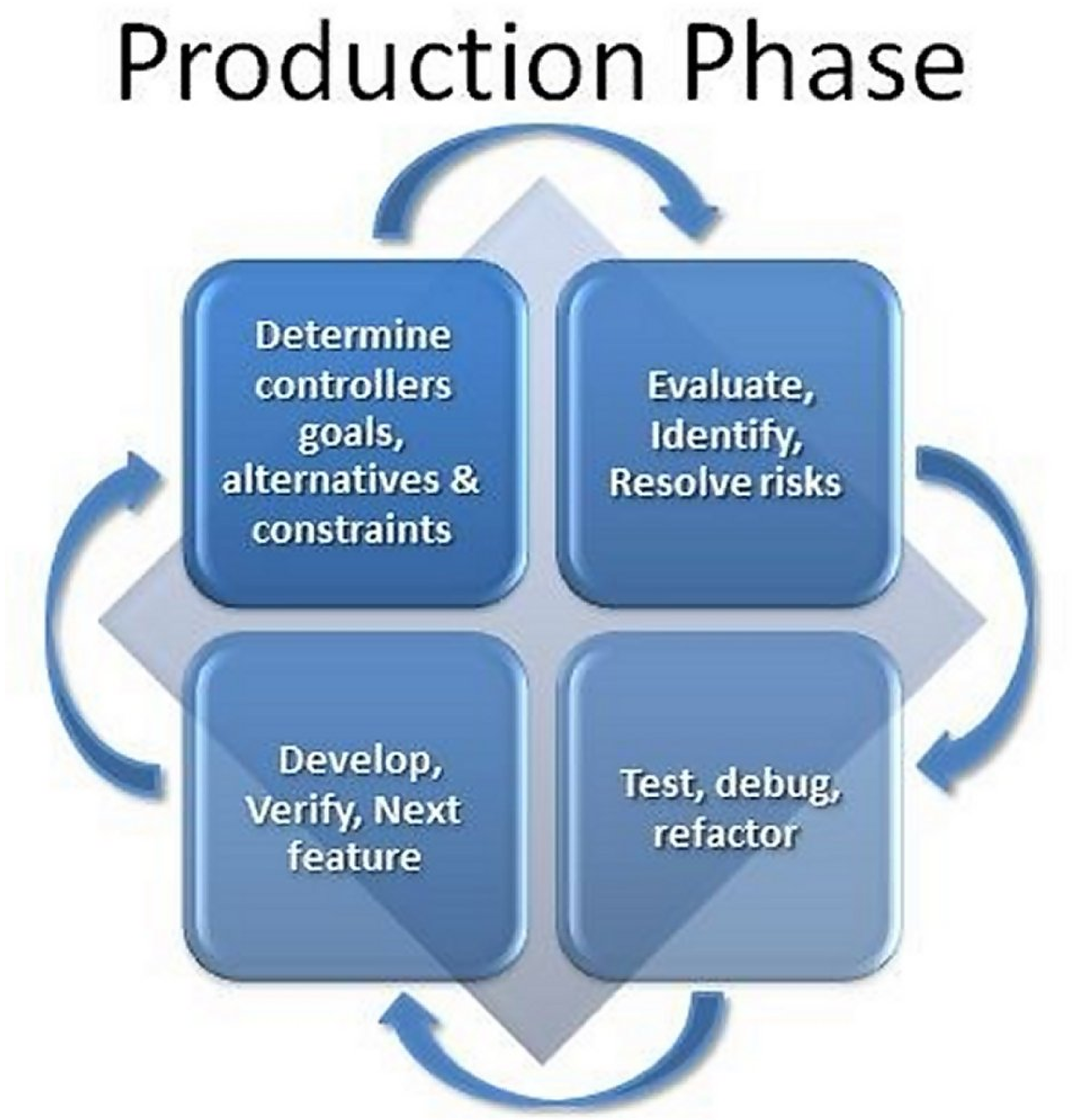
Situations where *least* appropriate:

1. Very small projects of very short duration.

2. Integration and architectural risks are very low.

3. Highly interactive applications where the data for the project already exists (completely or in part), and the project largely comprises analysis or reporting of the data.

Spiral


Production Phase
Determine controllers goals, alternatives & constraints
Evaluate, Identify, Resolve risks
Develop, Verify, Next feature
Test, debug, refactor

Framework Type: Combination Linear and Iterative

Basic Principles:

1. Focus is on risk assessment and on minimizing project risk by breaking a project into smaller segments and providing more ease-of-change during the development process, as well as providing the opportunity to evaluate risks and weigh consideration of the project continues throughout the life cycle.
2. “Each cycle involves a progression through the same sequence of

steps, for each portion of the product and for each of its levels of elaboration, from an overall concept-of-operation document down to the coding of each individual program." (Boehm, 1986)

3. Each trip around the spiral traverses four basic quadrants: (1) determine objectives, alternatives, and constraints of the iteration; (2) evaluate alternatives; identify and resolve risks; (3) develop and verify deliverables from the iteration; and (4) plan the next iteration. (Boehm, 1986 and 1988)

4. Begin each cycle with an identification of stakeholders and their win conditions, and end each cycle with review and commitment. (Boehm, 2000)

Strengths:

1. Enhances risk avoidance.

2. Useful in helping to select the best methodology to follow for development of a given software iteration, based on project risk.

3. Can incorporate Waterfall, Prototyping, and Incremental methodologies as special cases in the framework, and provide guidance as to which combination of these models best fits a given software iteration, based on the type of project risk. For example, a project with low risk of not meeting user requirements, but high risk of missing budget or schedule targets would essentially follow a linear Waterfall approach for a given software iteration. Conversely, if the risk factors were reversed, the Spiral methodology could yield an iterative Prototyping approach.

Weaknesses:

1. Challenging to determine the exact composition of development methodologies to use for each iteration around the Spiral.

2. Highly customized to each project, and therefore is quite complex, limiting reusability.

3. A skilled and experienced project manager is required to determine how to apply it to any given project.

4. There are no established controls for moving from one cycle to another cycle. Without controls, each cycle may generate more work for the next cycle.

5. There are no firm deadlines. Cycles continue with no clear termination condition, so there is an inherent risk of not meeting budget or schedule.

6. Possibility exists that project ends up implemented following a Waterfall framework

Situations where *most* appropriate:

1. Real-time or safety-critical systems.
2. Risk avoidance is a high priority.
3. Minimizing resource consumption is not an absolute priority.
4. The Project Manager is highly skilled and experienced.
5. A requirement exists for strong approval and documentation control.
6. The Project might benefit from a mix of other development methodologies.
7. A high degree of accuracy is essential.

8. Implementation has priority over functionality, which can be added in later versions.

Situations where *least* appropriate:

1. Risk avoidance is a low priority.
2. A high degree of accuracy is not essential.
3. Functionality has priority over implementation.
4. Minimizing resource consumption is an absolute priority.

RAD (Rapid Application Development)

Framework Type: Iterative

Basic Principles:

1. The key objective is for fast development and delivery of a high quality the system at a relatively low investment cost.

2. Attempts to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.

3. Aims to produce high-quality systems quickly, primarily through the use of iterative Prototyping (at any stage of development), active user involvement, and computerized development tools. These tools may include Graphical User Interface (GUI) builders, Computer Aided Software Engineering (CASE) tools, Database Management System (DBMS), fourth-generation programming languages, code generators, and object-oriented techniques.

4. Key emphasis is on fulfilling the business need, while technological or engineering excellence is of lesser importance.

5. Project control involves prioritizing development and defining delivery deadlines or “time boxes”. If the project starts to slip, emphasis is on reducing requirements to fit the time box, not in increasing the deadline.

6. Generally includes Joint Application Development (JAD), where users are intensely involved in system design, either through consensus building in structured workshops, or through electronically facilitated interaction.

7. Active user involvement is imperative.

8. Iteratively produces production software, as opposed to a throwaway prototype.

9. Produces documentation necessary to facilitate future development and maintenance.

10. Standard systems analysis and design techniques can be fitted into this framework.

Strengths:

1. The operational version of an application is available much earlier than with Waterfall, Incremental, or Spiral frameworks.

2. Because RAD produces systems more quickly and to a business focus, this approach tends to produce systems at a lower cost.

3. Engenders a greater level of commitment from stakeholders, both

business and technical, than Waterfall, incremental, or Spiral frameworks. Users are seen as gaining more of a sense of ownership of a system, while developers are seen as gaining more satisfaction from producing successful systems quickly.

4. Concentrates on essential system elements from user viewpoint.
5. Provides the ability to rapidly change system design as demanded by users.
6. Produces a tighter fit between user requirements and system specifications.
7. Generally produces dramatic savings in time, money, and man-hours.

Weaknesses:

1. More speed and lower cost may lead to lower overall system quality.
2. Danger of misalignment of developed system with the business due to missing information.
3. Project may end up with more requirements that needed (gold-plating).
4. Potential for feature creep where more and more features are added to the system course of development.
5. The potential for inconsistent designs within and across systems.
6. Potential for violation of programming standards related to inconsistent naming conventions and inconsistent documentation.
7. The difficulty with module reuse for future systems.
8. The potential for the designed system to lack scalability.
9. Potential for lack of attention to later system administration needs built into the system.
10. The high cost of commitment on the part of key user personnel.
11. Formal reviews and audits are more difficult to implement that for a complete system.
12. Tendency for difficult problems to be pushed to the future to demonstrate early success to senior management.
13. Since some modules will be completed much earlier than others, well-defined interfaces are required.

Situations where most appropriate:

1. Project is of small-to-medium scale and of short duration (no more than 6 man-years of development effort).
2. Project scope is focused, such that the business objectives are well defined and narrow.
3. Application is highly interactive, has a clearly defined user group, and is not computationally complex.
4. Functionality of the system is clearly visible at the user interface.

5. Users possess detailed knowledge of the application area.
6. Senior management commitment exists to ensure end-user involvement.
7. Requirements of the system are unknown or uncertain.
8. It is not possible to define requirements accurately ahead of time because the situation is new or the system being employed is highly innovative.
9. Team members are skilled both socially and in terms of business.
10. Team composition is stable; continuity of core development team can be maintained.
11. Effective project control is definitely available.
12. Developers are skilled in the use of advanced tools.
13. Data for the project already exists (completely or in part), and the project largely comprises analysis or reporting of the data.
14. Technical architecture is clearly defined.
15. Key technical components are in place and tested.
16. Technical requirements (e.g., response times, throughput, database sizes, etc.) are reasonable and well within the capabilities of the technology being used. Targeted performance should be less than 70% of the published limit of the technology.
17. Development team is empowered to make design decisions on a day-to-day basis without the need for consultation with their superiors and decisions can be made by a small number of people who are available and preferably co-located.

Situations where least appropriate:

1. Very large, infrastructure projects; particularly large, distributed information systems such as corporate-wide databases.
2. Real-time or safety-critical systems.
3. Computationally complex systems, where complex and voluminous data must be analyzed, designed, and created within the scope of the project.
4. Project scope is broad and the business objectives are obscure.
5. Applications in which the functional requirements have to be fully specified before any programs are written.
6. Many people must be involved in the decisions on the project, and the decision makers are not available on a timely basis or they are geographically dispersed.
7. The project team is large or there are multiple teams whose work needs to be coordinated.
8. When user resource and/or commitment is lacking.
9. There is no project champion at the required level to make things happen.
10. Many new technologies are to be introduced within the scope of the

the project, or the technical architecture is unclear and much of the technology will be used for the first time within the project.

11. Technical requirements (e.g., response times, throughput, database sizes, etc.) are tight for the equipment that is to be used.

Test Driven Development

Framework Type: Iterative

Basic Principles:

"Test-driven development" refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring).

It can be succinctly described by the following set of rules:

- write a "single" unit test describing an aspect of the program
- run the test, which should fail because the program lacks that feature
- write "just enough" code, the simplest possible, to make the test pass
- "refactor" the code until it conforms to the simplicity criteria
- repeat, "accumulating" unit tests over time

Expected Benefits

Many teams report significant reductions in defect rates, at the cost of a moderate increase in initial development effort the same teams tend to report that these overheads are more than offset by a reduction in effort in projects' final phases although empirical research has so far failed to confirm this, veteran practitioners report that TDD leads to improved design qualities in the code, and more generally a higher degree of "internal" or technical quality, for instance improving the metrics of cohesion and coupling

Common Pitfalls

Typical individual mistakes include:

- forgetting to run tests frequently
- writing too many tests at once
- writing tests that are too large or coarse-grained
- writing overly trivial tests, for instance omitting assertions
- writing tests for trivial code, for instance accessors

Typical team pitfalls include:

- partial adoption - only a few developers on the team use TDD
- poor maintenance of the test suite - most commonly leading to a test suite with a prohibitively long running time
- abandoned test suite (i.e. seldom or never run) - sometimes as a result of poor maintenance, sometimes as a result of team turnover

Signs of Use

- "Code coverage" is a common approach to evidencing the use of TDD; while high coverage does not guarantee appropriate use of TDD, coverage below 80% is likely to indicate deficiencies in a team's mastery of TDD
- version control logs should show that test code is checked in each time product code is checked in, in roughly comparable amounts

Skill Levels

Beginner

- able to write a unit test prior to writing the corresponding code
- able to write code sufficient to make a failing test pass

Intermediate

- practices "test driven bug fixing": when a defect is found, writes a test exposing the defect before correction

- able to decompose a compound program feature into a sequence of several unit tests to be written
- knows and can name a number of tactics to guide the writing of tests (for instance "when testing a recursive algorithm, first write a test for the recursion terminating case")
- able to factor out reusable elements from existing unit tests, yielding situation-specific testing tools

Advanced

- able to formulate a "roadmap" of planned unit tests for a macroscopic features (and to revise it as necessary)
- able to "test drive" a variety of design paradigms: object-oriented, functional, event-drive
- able to "test drive" a variety of technical domains: computation, user interfaces, persistent data access...

Further Reading on Test Driven Development

Test Driven Development: By Example, by Kent Beck

Appendix Foot Notes:

1. See Game Business Development Appendix & References

2. See Appendix for Game Design & References.

3. See article at Gamasutra* A Primer for the Design Process.

4. http://en.wikipedia.org/wiki/Systems_development_life_cycle

5. See article at: P4Connect Project Management Software into the Unity engine. P4Connect embeds the firm's P4D versioning engine into the Unity developer environment, allowing uses to access Perforce's features â€“ such as code and asset management, tracked change history and automation â€“ directly within Unity.

Developers will not need a Unity Pro or Team Unity licence, making P4Connect available to all indie devs that use Unity.

1. See Project Management Appendix: RAD step 3.