**Waterfall Model: *todas las demás son ágiles.***

While not as common in game development, some studios may use a structured waterfall approach for smaller or less complex games, with distinct phases like design, development, testing, and deployment.

**Agile Development:**

Many game development studios embrace Agile methodologies due to the iterative and flexible nature of game development. Frequent releases and adaptability to changing requirements are crucial in the dynamic gaming industry.

**Scrum:**

Scrum is widely used in game development, particularly for managing the production of specific features, levels, or game components within fixed-length sprints.

**Kanban:**

Kanban is applied in game development to visualize workflows, manage tasks, and optimize the development process, especially in live service games where continuous updates and content additions are common.

**Iterative Development:**

Game development often involves a process of iteration, where developers create a playable version of the game, gather feedback, and then iterate on the design, mechanics, and features based on that feedback.

**Rapid Prototyping:**

Game developers frequently use rapid prototyping to quickly build and test game concepts, mechanics, and ideas. This approach helps in identifying potential issues early in the development process.

**Scrum with Milestones:**

Some game development teams combine Scrum with milestone planning, where they set specific goals or checkpoints for major features or content releases, aligning with the overall development schedule.

**Feature-Driven Development (FDD):**

FDD can be applied in game development, especially for larger projects where breaking down features into manageable chunks is essential for efficient development.

**LiveOps Development:**

LiveOps is a methodology focused on ongoing operation and updates of games after launch. It involves continuous monitoring, analyzing player behavior, and delivering regular updates and new content.

**Extreme Programming (XP):**

XP practices, such as continuous testing and pair programming, can be adapted to game development, especially in scenarios where rapid development and frequent releases are crucial.

**Agile Game Development with DevOps:**

Combining Agile development with DevOps practices helps streamline the development pipeline, automate testing, and enable continuous integration and deployment for game updates.

**Software Methodology:**

* A methodology in software development refers to the overall approach or process used to plan, structure, and control the process of developing an information system or software application.
* It encompasses the principles, practices, and procedures followed during the software development lifecycle, guiding the team through different stages such as requirements gathering, design, implementation, testing, deployment, and maintenance.
* Examples of software methodologies include Agile, Scrum, Waterfall, Kanban, and others. These methodologies provide a set of guidelines for managing tasks, collaboration, and project delivery.

**Software Architecture:**

* Software architecture, on the other hand, focuses on the high-level structure and design of the software system. It defines the system's components or modules, their relationships, and the principles guiding their organization.
* Software architecture addresses concerns like scalability, maintainability, performance, and flexibility. It provides a blueprint for the overall system, guiding developers in making design decisions that align with the system's goals and requirements.
* Examples of software architectures include monolithic architecture, microservices architecture, client-server architecture, and more. These architectures dictate how the software components interact and are organized to achieve the desired functionality.

In summary, a software methodology is concerned with the overall process and management of the software development lifecycle, while software architecture focuses on the high-level design and structure of the software system. Both are essential aspects of successful software development, and they complement each other in guiding the development team through the creation of reliable, scalable, and maintainable software solutions.

**Entity-Component-System (ECS):**

A design pattern that separates entities (objects), components (features or behaviors), and systems (logic that processes components). ECS is known for its scalability and performance benefits.

**Model-View-Controller (MVC):**

Separates the game's internal representation (model) from how it is presented to the user (view) and how user input is handled (controller). This can help manage complexity in game development.

**Service-Oriented Architecture (SOA):**

Divides the game into independent, loosely coupled services that handle specific functionalities. Each service communicates with others through well-defined interfaces.

**Component-Based Architecture:**

Similar to ECS, this approach emphasizes building game entities as compositions of reusable and interchangeable components, allowing for flexibility and reusability.

**Game State Stack:**

Organizes different states of the game (menu, playing, paused) as a stack. Transitioning between states involves pushing and popping them from the stack.

**Finite State Machine (FSM):**

Represents different states of game entities and transitions between them based on certain conditions. FSMs are often used for character behavior, AI, and game logic.

**Client-Server Architecture:**

Commonly used in multiplayer games, where the game world and logic are separated between a client (player's device) and a server (manages game state, physics, etc.).

**Peer-to-Peer Architecture:**

Another multiplayer architecture where each player's device acts as both a client and a server, directly communicating with other players.

**Data-Driven Design:**

Employs external data files or scripts to drive game behavior, allowing for easier tweaking and modification without altering the code.

**Event-Driven Architecture:**

Game systems communicate through events, where actions or changes trigger events that are then handled by relevant systems or components.

**Layered Architecture:**

Separates different aspects of the game, such as rendering, physics, and AI, into layers with well-defined interfaces. Each layer focuses on a specific set of functionalities.

**Microservices Architecture:**

Similar to its use in general software development, microservices architecture can be applied to game development, especially in large-scale projects where different game features are treated as independent services.

These architectural patterns can be combined or adapted based on the specific needs of the game project. The choice of architecture depends on factors like the complexity of the game, the size of the development team, and the targeted platforms.