Usabilidad y Análisis de Juegos

Guillermo Jiménez Díaz (gjimenez@ucm.es)
Pilar Sancho Thomas (psancho@ucm.es)

Curso 2023-24

Tema 4. Diseño de un sistema de telemetría

El sistema de telemetría (sistema de *tracking* o **tracker**) es el responsable de recoger datos del juego mientras que éste está siendo jugado. Recopila valores de los atributos de cualquier objeto relacionado con el juego y aporta los *datos en crudo*, que posteriormente se convertirán en métricas.

La *instrumentalización* es la acción de añadir a nuestro videojuego código para recoger datos. Generalmente, la instrumentalización consiste en llamar a métodos del tracker para crear *eventos* (o trazas) en lugares específicos de nuestro código.

Para la recogida de eventos podemos utilizar servicios de terceros, ya sean genéricos (Firebase, Flurry, Oracle Analitycs Server...), dedicados a videojuegos (Azure Playfab, Gamesparks, Heroic labs, Game Analytics...) o específicos del motor (como Unity Analytics, o Unreal Analytics). También podemos implementar nuestros propios sistemas de telemetría. Algunos servicios de terceros tienen limitaciones en el número de eventos a enviar, formatos, parámetros, retardos en el acceso a las métricas... y la mayoría no nos darán acceso a nuestros datos en crudo (a no ser que paguemos).

A continuación vamos a dar algunas pautas a la hora de seleccionar, utilizar y diseñar un sistema de telemetría.

Funcionamiento básico

En general, un sistema de telemetría funciona de la siguiente forma:

- Se configura e inicia el tracker. Esto puede implicar comunicación con el servidor para solicitar información sobre atributos persistentes, qué tipos de eventos se van a recoger, etc. En este momento también se suelen guardar los primeros eventos

relacionados con el inicio de la sesión.
- Desde el videojuego se crean y mandan los eventos al tracker (*instrumentalización*).
- El tracker almacena internamente los eventos *en una cola de eventos*.
- Cada cierto tiempo, el tracker vuelca los eventos a disco o a un servidor (*persistencia*) en un determinado formato (*serialización*).
- Se finaliza la ejecución del tracker, guardando los eventos de finalización de sesión.

Características básicas

Sea un sistema de telemetría externo o desarrollado por nosotros mismos, es importante que cumpla con las siguientes características:

- Ha de ser independiente del videojuego y tener una base reutilizable para otros juegos.
- La recopilación de datos ha de producir el mínimo impacto posible sobre el videojuego.
- El volumen de datos ha de ser el adecuado (ni mucho ni poco) y teniendo en cuenta en qué condiciones puede ser usado (consolas, móviles, pruebas con usuario o uso real...).
- El formato de los eventos ha de ser simple y adecuado para ser convertido en métricas.
- En caso de desarrollo propio, se recomienda que sea extensible: que pueda ser acomodado a futuras mejoras, cambios de formatos, nuevos datos...

Diseño del sistema de telemetría

Como ya hemos estudiado en temas anteriores, el diseño del sistema de telemetría comienza planteando los **objetivos y las preguntas de investigación**, las cuales definen qué queremos analizar. Posteriormente, definiremos las **métricas** que usaremos en la evaluación y que servirán para contestar a las preguntas anteriormente definidas. Por último, definiremos los **eventos** que se utilizarán para el cómputo de las métricas propuestas. Éstos, además, servirán para diseñar correctamente el sistema de telemetría.

No seguir este proceso (objetivos-métricas-eventos) conduce a los problemas ya conocidos relativos a la analítica de juegos: pérdida de tiempo, de dinero, sobrecarga de datos e imposibilidad de obtener conclusiones relevantes.

A la hora de diseñar un tracker hay que tomar ciertas decisiones de diseño a priori:

- **Eventos**. Cuáles son los tipos de eventos que vamos a recopilar y cual es el formato/s en los que se van a almacenar.
- **Frecuencia**. Cada cuánto tiempo se va a recopilar cada tipo de evento. Existen eventos que se recopilan inmediatamente cuando se disparan pero otros, como la posición o el ancho de banda disponible, pueden ser recopilados cada cierto tiempo (*sampling*).
- **Almacenamiento**. Dónde se van a guardar los eventos, ya sea enviándolos a un servidor o guardándolos directamente en el equipo en el que se ejecuta el juego.
- **Configuración**. Decidir si el tracker siempre genera el mismo tipo de eventos o si puede ser configurado para activar o desactivar ciertos tipos de eventos en función de la build del juego. Así mismo, puede ser deseable decidir qué se va a hacer con los datos generados y si se van a enviar a uno o varios servidores distintos.

Al diseñarlo hay que tener en cuenta qué vamos a hacer cuando se producen ciertos *eventos catastróficos*:

- Nos hemos quedado sin espacio.
- No tenemos red.
- No tenemos permisos para almacenar los eventos.
- Consumo de batería y datos (móviles).
- El usuario reinstala la aplicación (dejamos de tener info del usuario concreto, si fuese necesario).
- El usuario modifica los datos enviados (poco probable en métricas).
- Privacidad en distintas regiones/países.

Instrumentalización

La instrumentalización establece cómo enviamos los eventos al tracker.

Solución *naive*: guardamos directamente en disco cuando toque.

```
// Aquí necesito guardar un evento
string miEvento = "timestamp,tipo,atrib1,atrib2";
File.WriteAllText(filename, miEvento);
// y espero a que abra y cierre el fichero
// y a que no falle
```

Solución *naive*: enviamos al servidor cuando toque.

```
// Aquí necesito guardar un evento
string miEvento = "timestamp,tipo,atrib1,atrib2";
```

```
RequestResponse response = IssueRequest( miEvento )
// y espero a que se comunique con el servidor
// y a que no falle la petición
// y a gestionar la respuesta del servidor
```

Como podremos imaginar, estas soluciones no son factibles. No puedo detener la ejecución del juego para enviar un evento. Además, esta solución es poco extensible y demasiado ad-hoc. Discutamos algunas soluciones mejores.

Punto de entrada

- Centralización del punto de entrada del sistema de telemetría en un objeto accesible desde cualquier punto de nuestro juego.
- Puede requerir de una inicialización y finalización explícitas.
- En la inicialización se pueden enviar eventos de inicio de sesión junto con parámetros que pueden aportar especificaciones adicionales: plataforma, SO, país, información demográfica (año de nacimiento, sexo, id de alguna red social...).

Envio de eventos al tracker

- Cadena formateada. Poco flexible

```
tracker.trackEvent("
    \"verb\": {
        "id": \"http://adlnet.gov/expapi/verbs/progressed\"
    }
    \"object\": {
        ...
    }
    ...
")
```

- Método "único": Distintas signaturas para un único método (con parámetros que pueden ser cadenas, enums, parámetros del evento, *argument objects*–diccionario, pares clave-valor)

```
tracker.trackEvent("Completable", "progressed", "Quest", "MyGameQuestId", progress)
```

- Métodos factoría (o *factory methods*): métodos que permiten crear un determinado tipo de evento que después es configurado antes de ser enviado. Útil si tenemos jerarquías de eventos

```
Event e = tracker.CreateCompletableQuestEvent("MyGameQuestId");
e.SetProgress(progress);
tracker.trackEvent(e);
)
```

- Fluent interface: se basa en realizar un encadenamiento de llamadas a métodos que hacen el código más legible. Implica un diseño de código más complejo para crear el lenguaje de dominio.

```
// Original
tracker.getCompletable().progressed(
                            "MyGameQuestId", CompletableTracker.Completable.Quest,
                            progress)

// Versión fluent
tracker.trackCompletable(
            tracker.Quest("MyGameQuestId")
                   .Progressed()
                   .withProgress(progress)
)
```

Eventos

Durante la instrumentalización, el programador indica al tracker cuándo y qué tipos de eventos se van a recoger. Además, añade toda la información necesaria que el evento ha de recoger (ver Figura 4.1).

Tipo

No todos los eventos son iguales por lo que es necesario que cada evento tenga un *tipo que lo identifique* (muerte, daño, recogida de objeto, inicio de nivel...). En este caso, es responsabilidad del programador decidir en cada momento cuál es el tipo del evento que se va a generar.

En general, hay una serie de eventos genéricos que suelen ser comunes a cualquier tipo de juego:

- Eventos de inicio de sesión: Eventos enviados cuando se ejecuta el videojuego en sí. Pueden ir integrados en el tracker, sin necesidad de instrumentalizar el código.

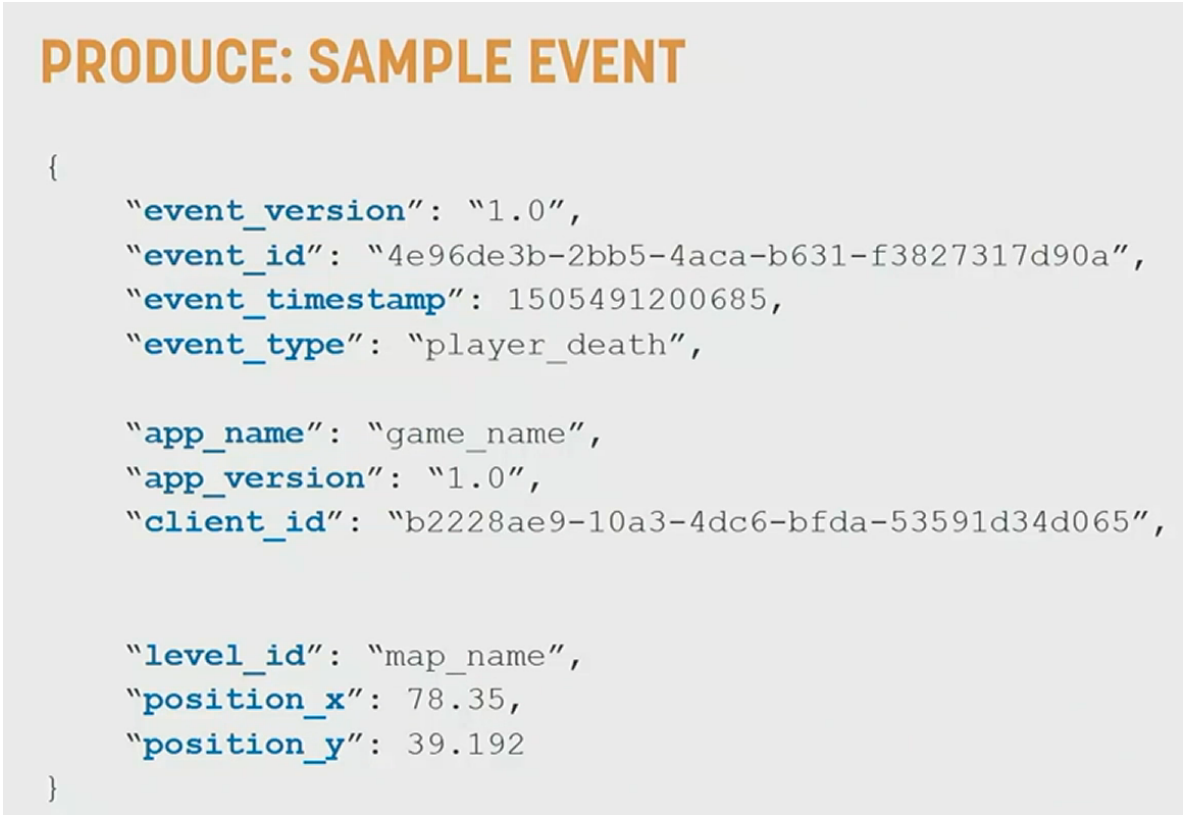

Figura 4.1: Ejemplo de evento serializado en formato JSON (Fuente: Building Scalable and Flexible Analytics Architectures for Games on AWS (GDC 2018))

Ej. Unity tiene *AppStart* para cuando se ejecuta el juego (o se trae a primer plano después de un periodo largo de inactividad) y *AppStop* cuando se detiene el juego. El programador no los ha de incluir sino que se envían por defecto.

- Eventos de progresión: Eventos enviados para medir la progresión del jugador en el juego: el jugador comienza una partida, el jugador la termina, el jugador comienza un nivel, el jugador abandona un nivel, el jugador termina el nivel (con un determinado resultado).

Ej. Unity tiene eventos de `game_start`, `game_over`, `level_start`, `level_complete`, `level_fail`, `level_quit`... Pueden enviarse un diccionario con parámetros adicionales para estos.

Ej. GameAnalytics tiene eventos `NewProgression(GAProgressionStatus, World, Level, Phase, resultado)`.

Ej. Tracker de uAdventure: `tracker.trackEvent(Çompletable", "progressed", "Quest", "MyGameQuestId", progress)`

El resto de eventos dependen de nuestro juego, tal y como vimos en un tema anterior.

Desde el punto de vista de la implementación, existen distintas formas de definir el tipo de los eventos:

- Números: ligero, fácil comparación, poco legibles. Necesaria "traducción" durante el análisis.
- Cadenas: pesado, comparación más pesada, muy legibles.
- Enum: Lo mejor de ambos mundos. Puede implicar métodos de conversión a cadenas (no todos los lenguajes lo soportan por defecto).
- Jerarquías de nombres (usado en GameAnalytics): El identificador de un evento es una cadena que representa un elemento dentro de una jerarquía de categorías y subcategorías. Esto facilita la organización de las métricas durante su análisis pero tiene las mismas desventajas que las cadenas. Ej:
  - `Tutorial:Step02:Start`: Comienza el segundo paso del tutorial.
  - `Sink:gold:boost:rainbowBoost`: el jugador ha gastado oro para comprar un rainbowBoost.
  - `Fail:PirateIsland:Sandyhills`: El jugador no ha completado el nivel SandyHills de PirateIsland.

Atributos comunes

Para cada evento es necesario definir qué información va a recoger, incluído el tipo del que hablamos anteriormente. Para ello, añadiremos una serie de atributos a cada uno de los eventos que vamos a generar. Aunque cada evento puede tener sus propios atributos, a continuación hablaremos más de los atributos fijos que todo evento ha de tener.

Además del tipo, todos los eventos deberían tener otros atributos fijos. En este caso, estos atributos pueden ser incorporados directamente por el propio tracker y no es necesario que el programador se encargue de añadirlos cuando instrumentaliza el código:

- timestamp (epoch o Tiempo POSIX) con zona horaria. No es recomendable enviar cadenas formateadas.
- id único de evento: para evitar duplicados.
- id sesión: arranque/cierre del juego, no distintas "partidas" sobre el mismo juego.
- id usuario o cuenta (si es un juego con login).
- id juego: Si somos capaces de hacer tracking de múltiples juegos a la vez.

Con respecto a la generación de ids (en caso de que no haya login), la mayoría de los motores ya proporcionan esa funcionalidad por defecto y crean un id de juego por cada instancia. Si es necesario generarlo, podemos crear cadenas únicas usando algoritmos como MD5 o SHA-1 (que es lo que usa Git para crear los identificadores de versión a partir del contenido de la copia de trabajo). Si queremos que sea variable, entonces es recomendable usar el timestamp más otra información única de la máquina (ejemplo: http://pid.github.io/puid/

Una última alternativa es que sea el servidor quien proporcione los ids de juego y de sesión al comenzar a trabajar con el tracker. Al arrancar una sesión de juego lo primero que hace el tracker es conectarse con el servidor para requerir esta información. A partir de entonces, todos los eventos llevan estos identificadores.

En algunos casos puede ser también necesario utilizar atributos de secuenciación, es decir, atributos que indican en qué orden se han producido los eventos. Estos atributos son necesarios porque tenemos que tener en cuenta que varios eventos pueden ser enviados durante el mismo frame (y, seguramente, con el mismo timestamp), pero que pueden llegar al servidor desordenados. Para el cálculo de algunas métricas, este desorden puede ser crítico e impedir un correcto cálculo de las mismas.

Eventos como objetos

En general, es recomendable que los eventos sean objetos y que, por tanto, definamos clases de eventos siguiendo el paradigma de la Orientación a Objetos. Esto se debe a que, en general, no van a ser procesados inmediatamente sino que serán almacenados por el sistema de telemetría en una *cola de eventos* antes de ser almacenados (ya sea en el dispositivo en el que se ejecuta el juego, ya sea en un servidor externo).

Si definimos los eventos como clases entonces tendremos una serie de ventajas:

- Podemos hacer el chequeo de la información que almacenan (validación).
- Podemos delegar en ellos ciertas responsabilidades como la *serialización*.
- Si los eventos solo contienen datos pueden implementarse como un *Bean* o *POJO*: un constructor sin argumentos, que tiene declarados todos sus atributos como privados y para cada uno de ellos un método setter y getter. Esto facilita el uso de algunas librerías de serialización de terceros.

Eventos muestreables

Cuando queremos que un evento no se envíe siempre que ocurra (p.e. posición del jugador en cada frame) sino cada cierto tiempo, tenemos que decidir cómo vamos a realizar el muestreo de dicho evento:

- Enviamos siempre el evento pero es el tracker quien decide cuándo lo almacena. Implica generar una llamada al tracker en cada tick que será ignorado en la mayoría de las ocasiones.
- Creamos un objeto monitor encargado de las propiedades que queremos capturar cada cierto tiempo. Este objeto se responsabiliza de hacer polling sobre las propiedades que quiere seguir. Implica un fuerte acoplamiento con el objeto que va a seguir (similar a lo que hacía el `AnalyticsEventTracker` modo Timer de Unity 2021)

Volcado

Como ya hemos dicho, los eventos se almacenan en una cola que, cada cierto tiempo, se guarda en disco, ya sea en un servidor, ya sea en el dispositivo en el que se esté ejecutando el juego, en un determinado formato.

Un buen sistema de tracking ha de separar los dos mecanismos de los que se compone el volcado:

- **Persistencia**: Cuál es el mecanismo usado por el sistema de telemetría para almacenar los datos generados por los eventos (se guardan a disco, se envían por red, se harán ambos...). Esto incluye decidir con qué frecuencia se van a persistir los datos y la inicialización de la capa de persistencia, si fuese necesario.
- **Serialización**: Cuál es el formato en el que se van a guardar los eventos.

Es recomendable mantener ambos subsistemas separados para evitar acoplamientos entre ambos y permitir combinaciones de unos con otros (o la coexistencia de varios de ellos).

Persistencia

El sistema de persistencia es el responsable de decidir qué hacer con los eventos que llegan al sistema de tracking. Como dijimos anteriormente, los eventos se suelen almacenar en una cola de eventos. Cada cierto tiempo es necesario hacer un volcado (*flush*) de esta cola y persistir los datos de los eventos.

Por tanto, se ha de definir cuál es la frecuencia con la que queremos que el sistema de telemetría vuelque a disco y/o envíe los eventos al servidor. Existen distintas posibilidades.

- Solo cuando la aplicación se pausa/termina. Evita retardos durante el juego pero pueden sobrecargar la memoria y producir retardos importantes entre sesiones de juego.
- Mediante temporizadores: Los eventos se van guardando en memoria y, cada cierto tiempo, se envían/guardan. Un mal uso de la temporización puede producir los mismos errores que el anterior.
- Mediante checkpoints: Los eventos se almacenan y se envían solo en momentos puntuales (finalización de un nivel, puntos concretos dentro del juego, etc).
- De manera individual (no streaming): no es necesario una cola de eventos sino que el tracker persiste cada evento según llega. Puede producir una sobrecarga importante por la gestión de la entrada/salida o la red.
- Streaming data: Se pueden usar infraestructuras en las que se cree una conexión punto a punto entre el juego y el servidor que permitan un envío continuo de datos. Suele ser usado en consolas y servicios asociados a la misma y se puede usar este mismo flujo para el envío de eventos en el mismo momento en que ocurran. Esto implica que la persistencia de los eventos es *síncrona*, es decir, cada vez que llega un evento, éste es serializado y almacenado.

Si se hace persistencia asíncrona, se recomienda hacer el volcado usando una hebra de

ejecución diferente, lo que obliga a que la cola pueda ser utilizada concurrentemente (para meter eventos y para sacarlos y realizar el volcado).

Así mismo, se pueden definir sistemas de telemetría complejos que durante el inicio de la sesión permitan configurar desde el servidor, entre otros, la frecuencia de muestreo o el tipo de eventos que se van a recoger. Esto evita realizar modificaciones en el propio juego.

Si cada sistema de persistencia es responsabilidad de un objeto entonces podemos hacer que un tracker pueda tener distintas estrategias de persistencia ejecutándose simultáneamente (por ejemplo, enviando a disco y a un servidor). Esto se puede conseguir mediante un patrón *Strategy*, que permite la convivencia de distintos algoritmos para la realización de una misma tarea.

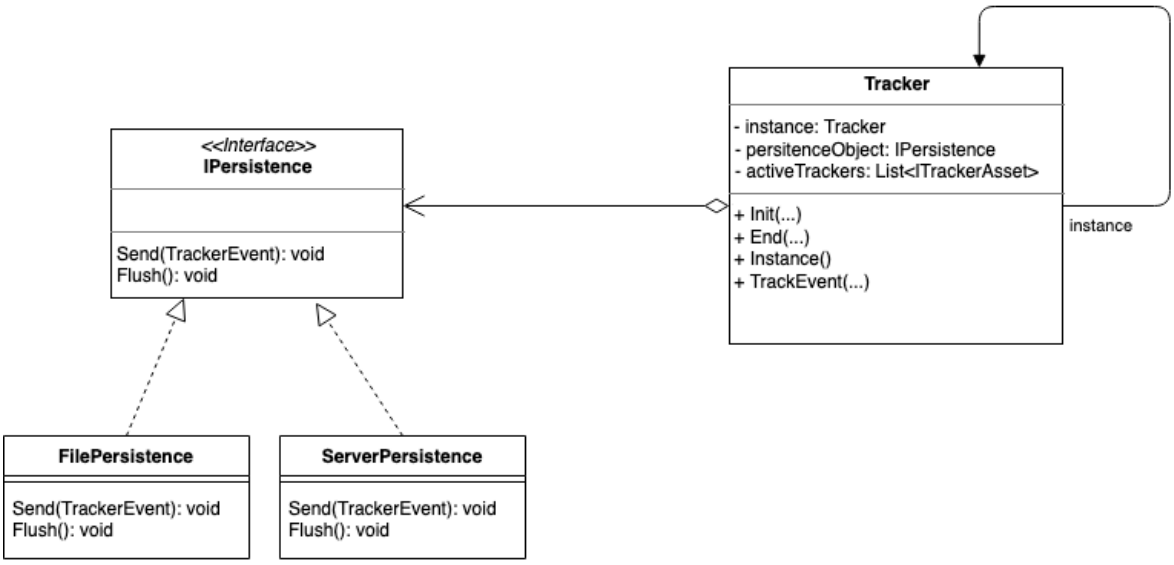

Figura 4.2: Uso de estrategias para la persistencia

Serialización

La serialización transforma los eventos a datos en un formato concreto. Estos datos serán persistidos por el sistema de persistencia.

Es importante decidir en la fase de diseño cuál es el formato en el que se van a enviar y guardar los eventos. Esta decisión de diseño afecta a cómo se procesan los datos, a la

sobrecarga en memoria y en red que pueden producir, etc. Algunos formatos comúnmente usados son:

- CSV: Fichero separado por comas. Es un formato "semilegible" y ligero, pero que no permite estructuras complejas.
- XML: Es un formato estructurado y legible, que permite guardar estructuras complejas de eventos. Sin embargo, es pesado.
- YAML: Es un formato usado, por ejemplo, por Unity en las escenas y demás información sobre assets. Es legible y permite estructuras complejas. Es más ligero que XML pero como conlleva indentación por espacios, puede ser propenso a errores.
- JSON: Es un formato legible y que permite estructuras complejas de eventos. Es muy usado por distintos trackers de terceros (como el de Unity o el de GameAnalytics). Puede ser aún más ligero que YAML y es fácilmente comprimible (*minimifiers*). Hay que tener cuidado con el uso de distintas librerías en los distintos lugares en los que se trabaje con los eventos (SimpleJson, fastJson, librerías propias del lenguaje...) ya que pueden producirse errores.
- Binary formats: No son legibles por humanos de manera directa pero ocupan menor tamaño y tienen un menor consumo de ancho de banda. Requieren de herramientas específicas de serialización y deserialización. Ej. Protocol buffer, un mecanismo de serialización de estructuras de Google para múltiples lenguajes, o Bond, la alternativa de Microsoft.

Una solución sencilla si permitimos distintos formatos de serialización sería incluir en todos los sistemas de persistencia código como este:

```
switch (format)
{
    case TraceFormats.json:
        queue.Add(event.ToJson());
        break;
    case TraceFormats.xml:
        queue.Add(event.ToXml());
        break;
    default:
        queue.Add(event.ToCsv());
        break;
}
```

El problema de esta solución es que acopla el sistema de persistencia con la serialización,

de modo que añadir un nueva forma de serialización implica modificar todos los sistemas de persistencia ya implementados.

Para mejorarlo, del mismo modo que hicimos con el sistema de persistencia en el tracker, las diferentes estrategias de serialización pueden ser objetos, de modo que un sistema de persistencia pueda delegar en uno o varios de ellos para generar los datos de los eventos en el formato adecuado.

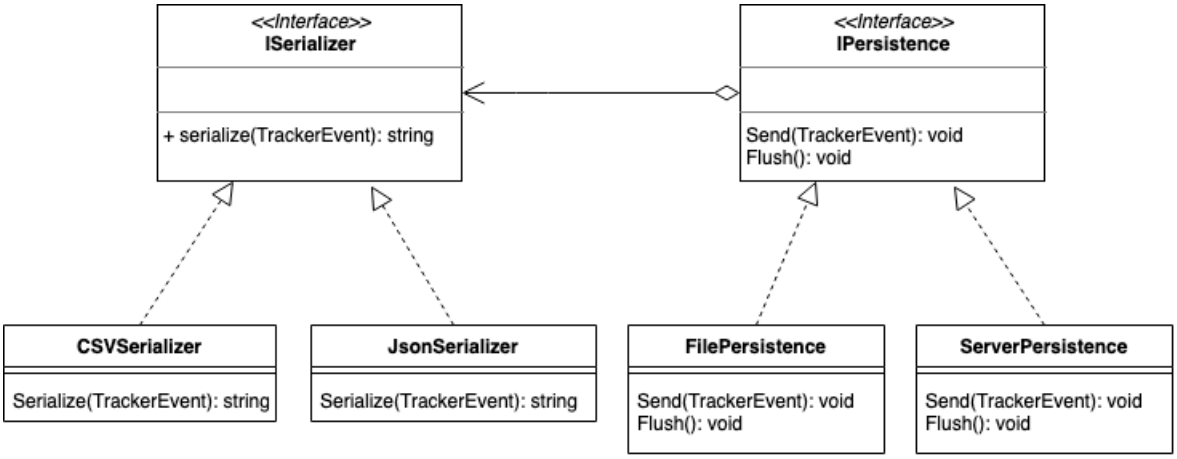

Figura 4.3: Uso de estrategias para serialización y persistencia

Desactivación de tracking de eventos

Puede ser aconsejable activar/desactivar el sistema de tracking de ciertos eventos en determinadas fases del desarrollo del videojuego:

- Por protección de datos.
- Porque hay eventos que son innecesarios en producción.
- Porque queremos realizas distintos tipos de pruebas en las que ciertos eventos son innecesario.
- ...

En ningún caso, la desactivación del tracking de cierto tipo de eventos puede suponer un fallo en el videojuego. Así mismo, se espera que estos cambios no impliquen, en la medida de lo posible, modificar la instrumentalización del juego (y, por tanto, crear una nueva release).

Para poder hacer esto, es necesario que el sistema de tracking permita gestionar qué tipos de eventos van a ser recogidos en cada momento. Algunas alternativas de diseño son:

- Definir diccionarios con los tipos de eventos a recoger.
- Uso de máscaras de bits para decidir el tipo de evento que se va a recoger.
- Trackers especializados: el sistema de tracking dispone de múltiples trackers, cada uno de los cuales procesa un determinado tipo de eventos. Cuando llega un evento se pregunta a cada uno de los trackers activos si lo acepta. Si ninguno lo acepta, se rechaza.

En todo caso, la configuración de qué tipos de eventos se van a recoger ha de ser externa al juego o al sistema de tracking, ya sea con ficheros de configuración, ya sea mediante comunicación con un servidor (handshake inicial).

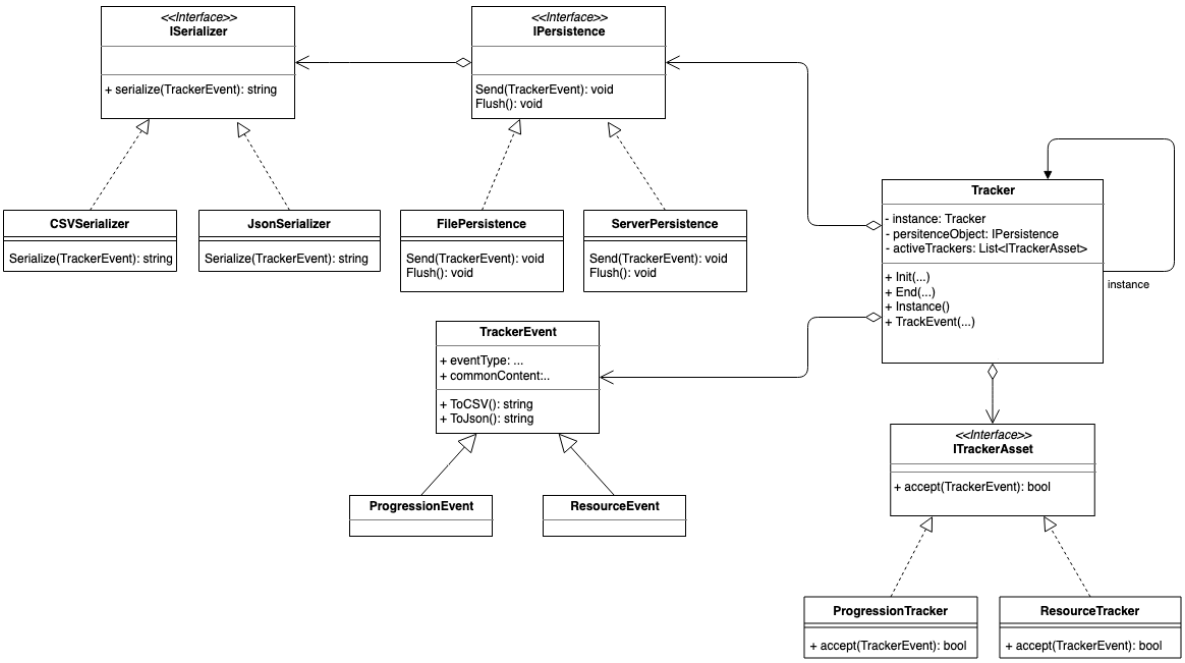

Figura 4.4: Sistema de telemetría con trackers especializados

Decisiones de diseño avanzadas

Algunas decisiones y mejoras que se pueden tener en cuenta en los sistemas de telemetría son las siguientes:

- Usar algoritmos como SHA-1 o MD5 para verificar la integridad de los datos al almacenarlos.
- Realizar comprobaciones para verificar que los datos que llegan al servidor siguiendo un determinado modelo de datos. Esto sirve principalmente si el sistema de telemetría ha pasado por distintas versiones de desarrollo.
- El uso de colas circulares, con el fin de controlar el exceso de eventos y evitar la sobrecarga de memoria.
- Uso de múltiples backups, ya que sobreescribir datos puede ser peligroso.
- Gestión de red, saturación del servidor o llenado de disco: decidir qué políticas se van a usar en caso de que no sea posible realizar la persistencia de los datos. Puede ser recomendable utilizar sistemas de persistencia híbridos inteligentes que decidan qué hacer con los eventos cuando no es posible conectar con el servidor.

Ética y privacidad

La privacidad el usuario es un aspecto clave. Podemos grabar datos muy detallados que pueden infringir la legalidad.

Actualmente la industria mantiene estos datos de forma confidencial, aunque podrían venderse a terceros al igual que en el comercio electrónico. Normalmente los datos se anonimizan.

Debemos informar a los usuarios del uso de los datos recogidos.

Web Analyst Code of Ethics (http://www.digitalanalyticsassociation.org/codeofethics)

Ejemplos y referencias usadas

- Webtics: https://github.com/SimonMcCallum/WebTics Cap. 10 del libro Game Analytics (en la bibliografía).
- Tracker eAdventure: https://github.com/e-ucm/rage-analytics/wiki/Tracker
- SnowPlow. Charla en la GDC 17 sobre este tracker: Open Source Game Analytics Powered by AWS
- GameAnalytics: Documentación
- Unity Analytics: Documentación
- Unreal Analtics: Documentación
- Game Analytics. Maximizing the Value of Player Data. Seif El-Nasr, Magy, Drachen, Anders, Canossa, Alessandro. Springer 2013.

- Unite 2016 - Best Practices in Persisting Player Data on Mobile
- Making “Big Data” Work for Halo: A Case Study
- Building Scalable and Flexible Analytics Architectures for Games on AWS. GDC 2018

Nota final