

Usabilidad y Análisis de Juegos

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

Curso 2023-24

Tema 6: Quality Engineering (DevQA) y automatización de pruebas

El objetivo de las pruebas es encontrar el mayor número posible de errores con una cantidad razonable de esfuerzo, aplicado sobre un lapso de tiempo realista. Para ello, en ocasiones podemos desarrollar pruebas que sean ejecutadas por un ordenador, en lugar de ser realizadas por un humano.

En el capítulo anterior hablamos de QA para referirnos a todas aquellas pruebas que hacen los probadores humanos para encontrar errores en un videojuego, en este capítulo, hablaremos del *Quality Engineering* o *DevQA*, es decir, en aquellas pruebas que no son realizadas por humanos sino que son ejecutadas usando herramientas de automatización de pruebas. No sustituyen las pruebas de QA pero reducen el trabajo que ha de hacer a los probadores (que se encargarán de las pruebas más creativas y menos automatizables).

Testing en videojuegos

En el desarrollo de software en general, no todas las pruebas automáticas tienen el mismo coste y se suelen representar como una pirámide (Figura 6.1) en la que en la parte superior se encuentran las pruebas que tienen un mayor coste y que, presumiblemente, serán realizadas por probadores humanos (que es un recurso de alto coste).

Tradicionalmente, las pruebas de videojuegos han recaído principalmente en probadores humanos debido a la complejidad del sistema software que es en sí mismo un videojuego, consiguiendo una pirámide invertida (Figura 6.2 izquierda).

Sin embargo, esta tendencia se está invirtiendo (Figura 6.2 derecha) gracias al uso masivo de pruebas automáticas. Se está viendo que existen una gran cantidad de errores que

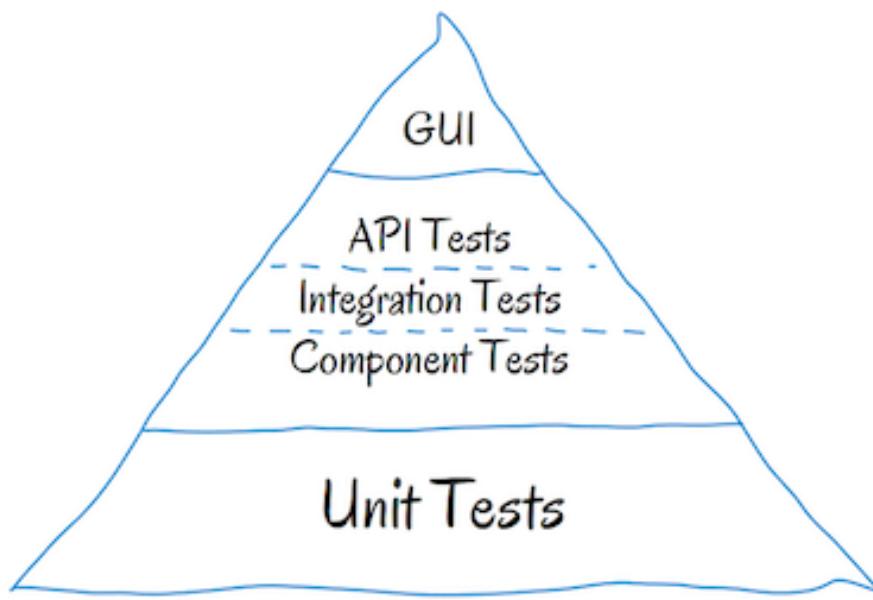


Figura 6.1: Pirámide de testing para pruebas automáticas (genérico) (Fuente: Agile game development with Scrum)

pueden ser corregidos en fases tempranas del desarrollo con pruebas sencillas, rápidas y de bajo coste, dejando que las más costosas (incluidas las de los probadores) solo se realicen en momentos más avanzados, en los que se está seguro de que las builds utilizadas son lo suficientemente estables.

Como veremos más adelante, las pruebas automáticas pueden ser de diversos tipos (unitarias, funcionales, de integración...). Así mismo, las pruebas no solo se refieren a código sino que también se pueden aplicar a los contenidos del juego (lightmaps, assets, datos de usuario almacenados por el juego...) o de otros sistemas que intervienen en el desarrollo del propio videojuego (como las pruebas de comunicación y de estrés sobre los servidores en videojuegos multijugador).

En todo caso, los sistemas de pruebas han de estar desarrollados de modo que sea más difícil no realizar las pruebas que realizarlas. Una de las ventajas principales de garantizar que se puedan ejecutar fácilmente es que vamos a poder ejecutarlos constantemente, teniendo un mayor control de calidad del software que estamos desarrollando. Por tanto, el coste de desarrollarlos (tanto los tests como la infraestructura para ellos) compensa con respecto al tiempo que tardaríamos en pasarlo con humanos y por la cantidad de veces que los podemos ejecutar.

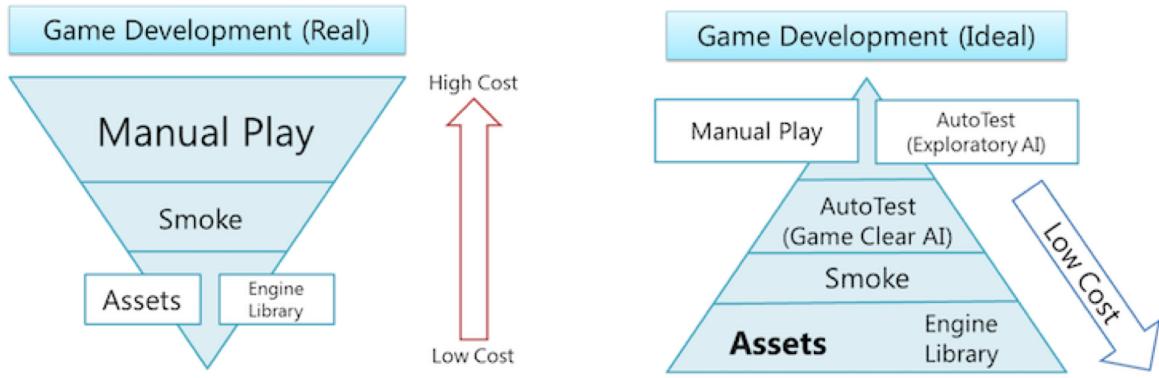


Figura 6.2: Pirámide de testing automático para videojuegos. A la izquierda, las pruebas que se hacían tradicionalmente; a la derecha, el nuevo panorama en el ámbito del testing(Fuente: [Sega Techblog](#))

Pruebas según la forma en la que se usan

Pruebas de caja blanca

Se centran en los detalles de implementación del software, por lo que su diseño está fuertemente ligado al código fuente. Ej: pruebas sobre componentes de salud, clases de la lógica del juego, motor de render...

El responsable de crear este tipo de pruebas ha de saber cómo está implementado el juego. Como tienen en cuenta la implementación entonces se encargan de probar:

- Todos los caminos independientes de cada módulo (componentes, clases, motores...).
- Todas las decisiones lógicas en su vertientes verdadera y falsa. – Todos los bucles en sus límites. – Las estructuras internas de datos.

Pruebas de caja negra

Las pruebas de caja negra se centran en la funcionalidad o jugabilidad del videojuego. Tienen en cuenta unas condiciones de entrada y se verifica que se cumple con la respuesta esperada. Por ejemplo: menús de selección y uso de botones, el jugador recibe daño cuando el enemigo explota cerca de él...

Desde el punto de vista del desarrollo, las pruebas de caja negra pueden ser usadas para comprobar la comunicación entre subsistemas, ya que no es necesario saber cómo están

implementados sino solo qué funcionalidades aportan.

El responsable de crear este tipo de pruebas ha de saber cómo se juega al juego.

Tipos de pruebas

Pruebas unitarias

Consiste en el desarrollo de pruebas (o *test de unidad*) para cada unidad funcional (clases) creada en nuestro software. Para cada unidad funcional (clase) hemos de probar cada una de sus operaciones (métodos) usando distintos casos. Debido al polimorfismo, las operaciones pueden variar en el contexto de cada clase.

Cada test de unidad prueba de manera única un método, pasándole unos parámetros de entrada y comprobando que la salida que produce es la esperada para dicha entrada. Los test de unidad han de probar tanto casos correctos como incorrectos y han de verificar convenientemente las condiciones límite. En general, no deben hacer uso de archivos para realizar configuraciones ni deberían de depender de otros métodos (algo difícil de conseguir).

Los tests de unidad usan **asertos**, funciones que comprueban que una variable cumple una determinada condición (valor esperado, valor observado). Muchos lenguajes dan soporte a los asertos, que sirven para sustituir al uso de excepciones (para hacer comprobaciones en runtime que solo son interesantes durante el desarrollo)

Para hacer los test se suele seguir lo que se conoce como el **patrón AAA** (*Arrange-Act-Assert*) que consiste en:

1. *Arrange*: Configuramos el estado de un objeto/función como nos interesa para hacer el test.
2. *Act*: Ejecutamos el método que tenemos que probar y guardamos la información que necesitamos comprobar que es correcta.
3. *Assert*: Verificamos que la información recopilada en el paso anterior es correcta con respecto a la ejecución esperada.

Un software puede terminar teniendo miles de test de unidad por lo que no es lógico que los test se prueben a mano. En general, todos los lenguajes de programación disponen de librerías que ayudan al desarrollo y a la ejecución automática de los test de unidad. Por ejemplo, para C# (usado en el Grado en primero) están NUnit, XUnit o MSUnit, mientras que para C++ (que se usa en el Grado en segundo) hay otras como Boost.Test o Google Test.

Pruebas de integración

Son pruebas para verificar que los módulos que componen nuestro juego (clases, componentes, subsistemas) son capaces de interactuar correctamente entre sí.

Después de probar las clases independientemente mediante los test de unidad se continúa una secuencia de pruebas por capas de las clases dependientes hasta que se construye el sistema completo.

Pruebas de sistema

Son pruebas de caja negra que verifican que el sistema completo se comporta como se espera de acuerdo a como ha sido definido (GDD o documentos de diseño y casos de uso, en aplicaciones convencionales). Ya no se tiene en cuenta en ningún caso que el videojuego está compuesto de distintos módulos sino que se trata como un sistema completo. Verifica que las funcionalidades son correctas para distintas entradas. En muchas ocasiones, las pruebas de sistema incluyen pruebas no funcionales que comprueban los tiempos de respuesta, instalación, estabilidad, rendimiento o escalabilidad, entre otros.

Estas pruebas suelen estar relacionadas con las **pruebas de aceptación**, que son las pruebas que hay que realizar para decidir que una aplicación está lista para ser lanzada.

End to end testing

Generalmente, las pruebas de sistema se concentran en las acciones visibles para el jugador y en la salida que el jugador puede reconocer. El *End to end testing* es similar a las pruebas de sistema pero con la salvedad de que tienen en cuenta cualquier otro subsistema del que dependa nuestro juego (por ejemplo, APIs de logros, sistemas de matchmaking...).

Verifica que el flujo de ejecución es el correcto de principio a fin. No suele abarcar todo el juego sino solo procesos muy concretos (una quest, conexión a una partida multijugador...)

Pruebas de regresión

Consisten en volver a ejecutar periódicamente un subconjunto de casos de prueba (puede incluir tests de unidad) para asegurar que los cambios nuevos introducidos (y probados independientemente) no generen efectos colaterales no deseados y que los errores ya resueltos no reaparezcan.

Revisan que no se hayan perdido mecánicas o que no aparezcan de nuevo errores que ya fueron corregidos. Se suelen producir cuando una nueva funcionalidad entra en colisión con una antigua previamente probada y verificada.

Dependiendo del coste/riesgo pueden ser más superficiales, probando solo los casos positivos. Aseguran que una funcionalidad previamente desarrollada aún es soportada por el sistema.

Smoke tests

Son revisiones “rápidas” y superficiales de que el juego funciona correctamente desde el punto de vista del usuario y que no se producen *crashes*. Prueban la creación de una build y que las funcionalidades principales estén funcionando correctamente. Estos tests han de poder ser ejecutados en un periodo de tiempo corto ya que se suelen ejecutar frecuentemente (por ejemplo, antes de subir una versión a un sistema de control de versiones).

Estas pruebas se pueden usar para evitar que una build que no funciona intente ejecutar otras pruebas automáticas que consumen más tiempo, para evitar que un probador humano pierda el tiempo trabajando con una build inestable o para hacer verificaciones finales antes de una release.

Pruebas de estrés y de carga

Se suelen ejecutar en aplicaciones que hacen uso de la red (como los videojuegos multijugador). Consisten en poner a prueba los servidores haciendo múltiples peticiones o desconectando ciertos servidores, en caso de que se tengamos sistemas distribuidos y queramos comprobar que los sistemas de equilibrado de carga funcionan correctamente.

Asset testing o pruebas de contenido

No son pruebas funcionales sino de los contenidos usados por el juego. Estas pruebas comprueban que los assets del videojuego (modelos, texturas, animaciones, sprites, iconos, archivos de datos...) están en el lugar esperado y son correctos.

Tests destructivos

Aquellas pruebas que intentan causar que el sistema falle de manera intencionada. Comprueban que el sistema sigue funcionando incluso cuando recibe inputs inesperados, haciendo que el sistema sea más robusto y estable.

Un tipo particular de tests destructivos es el **Monkey testing**. Consiste en probar una aplicación generando inputs aleatorios y comprobando que la aplicación funciona correctamente (o que genera errores). Son tests generalmente rápidos de implementar y fácilmente automatizables ya que no definen casos de pruebas ni una estrategia concreta. Además, pueden ser capaces de detectar errores completamente inesperados. Sin embargo, al ser inputs aleatorios, da lugar a que se encuentren errores que no son reproducibles. Además, suelen consumir mucho tiempo (es necesario tenerlos ejecutándose durante bastante tiempo para conseguir resultados interesantes).

En Netflix tienen un **ejército de simios** e, incluso, un Chaos Kong (el gorila más destructor).

Flaky tests

No son en sí un tipo de pruebas sino que representan a aquellos tests que reportan errores que pueden ocurrir o no para la misma configuración. Esto implica que un bug no siempre significa un error en el código. Se pueden originar por:

- Concurrencia: los tests de integración se suelen ejecutar en paralelo y puede ocurrir que la ejecución en una hebra pueda provocar un error en otra.
- Caches de datos
- Setup and clean states: si se nos olvida limpiar lo que ha generado un test esto puede provocar efectos laterales en otros tests.
- Contenido dinámico que requiera carga o ejecución asíncrona: los tests siempre se ejecutarán más rápido que como lo realiza un humano.
- Bombas de tiempo: relacionados con peticiones de cuál es el momento actual (y en qué zona horaria estamos). También está relacionado con el uso de intervalos de tiempo, ticks de simulación, el momento del día en el que se ejecuta (como **el misterioso error de “Little Big Planet, Eye Toy y la aspiradora”**)
- Infraestructura: Los tests fallan por motivos externos (latencia en la red, versión del navegador, una consola ha fallado, fallos en nodos en sistemas de integración continua distribuidos...)
- 3rd parties: falla la integración con sistemas de terceros, no con los propios

En general, para afrontar estos errores se suelen usar estas soluciones:

- Ejecutar los tests varias veces antes de darlos definitivamente por fallos. Cuidado porque parece como que cualquier error puede ser debido a fallos de terceros y que algunos tests pueden tardar demasiado en volver a ser ejecutados.
- Poner errores/tests en cuarentena e investigarlos más detenidamente.

- Ejecutar tests de manera planificada y frecuentemente, a distintas horas del día, junto con los informes de tests, ayuda a encontrar este tipo de errores.

Sistemas de integración continua

La integración continua consiste en la verificación constante de que el software desarrollado por todos los desarrolladores se integra (funciona) correctamente. Este proceso se realiza de manera automática mediante los *servidores de integración continua*, que se instalan asociados al sistema de control de versiones.

El flujo de trabajo en la integración continua es el siguiente (Figura 6.3): Por cada commit (o periódicamente), el servidor de integración continua se descarga todo el código del repositorio del sistema de control de versiones, lo compila (crea una build completa) y ejecuta todas las pruebas (de unidad, integración, validación...). Si los test se pasan entonces la build que hay en el repositorio es correcta y se puede continuar trabajando. Sin embargo, si la build falla, el desarrollo se detiene y se identifica a los responsables para que lo arreglen cuanto antes para evitar que el error se propague. Esto implica que los desarrolladores deberán trabajar en ramas cortas (*trunk-based development*) en lugar de desarrollar con ramas largas.

Los sistemas de integración continua pueden añadir todo tipo de tareas adicionales, como la compilación en varias plataformas, la ejecución de pruebas de testeo y de integración, pruebas de contenidos, verificación de nombres de archivos, guardado de información para la generación de métricas de rendimiento, envío periódico de builds al equipo de QA, etc. En estos casos se suele hablar de Continuous Delivery (en lugar de Continuous Integration).

Algunos ejemplos de este tipo de sistemas son: Jenkins, TeamCity, Github Actions, Travis CI, Circle CI, GitLab CI o Cruise Control, entre otros. Como veremos en algún ejemplo de la industria, algunas empresas han desarrollado sus propios sistemas de integración continua.

Pruebas automatizadas o *automated testing*

The difference between us and a computer is that, the computer is blindingly stupid, but it is capable of being stupid many, many million times a second.

Douglas Adams (SCO Forum 97)

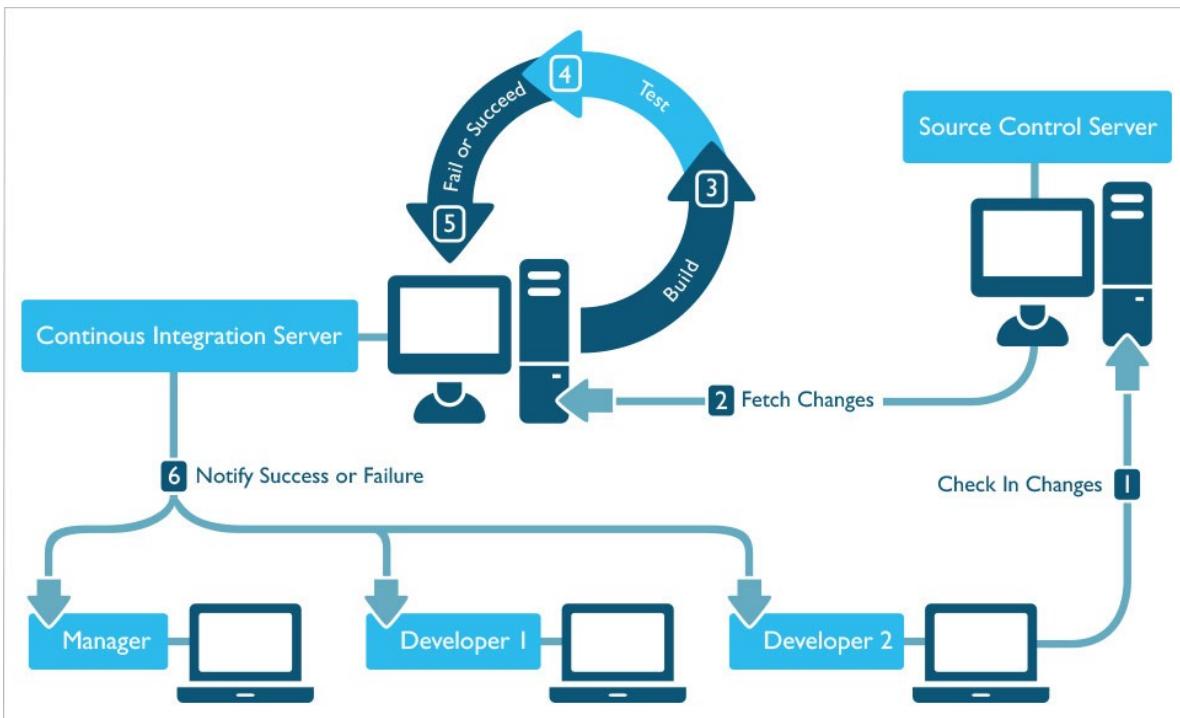


Figura 6.3: Flujo de trabajo de los sistemas de Integración Continua

La potencia de los test automáticos se encuentra en la capacidad de poder repetir una cantidad grande de pruebas de manera sencilla y medible. Los sistemas de pruebas automáticos no solo se encargarán de ejecutar los tests sino también de generar datos con los que hacer un seguimiento del desarrollo. Los datos generados por estos sistemas se utilizan para generar alarmas, elaborar métricas del progreso de desarrollo, generar informes de seguimiento y capturar otro tipo de información relacionada con el rendimiento del videojuego o que ha de ser procesada por humanos.

En ocasiones veremos que los tests automáticos surgen como respuesta a tests realizados por humanos (responsabilidad de los QA engineers). Es necesario que las pruebas automáticas sean escalables y reproducibles (en la medida de lo posible).

Hay tres estrategias fundamentales a la hora de crear test automáticos:

- Scripting: Los responsables de QA, junto con los desarrolladores, crean los comportamientos que van a servir para testear el código.
- Exploración: Se crean bots que se dedican a explorar y a realizar acciones de manera más o menos aleatoria/inteligente sobre el mundo de juego.
- Replay: Se guardan las trazas de jugadores humanos (input y estado de juego) con el fin de que se puedan reproducir de manera automática lo que hizo el jugador humano.

Recordemos que, aunque sean automáticos, todos los tests tienen un coste y hay que saber cómo y cuándo combinar los tests automáticos y los tests con humanos (Figura 6.4):

- Usaremos los tests “baratos” para encontrar errores de grano grueso. Se ejecutan rápidamente y permiten que el resto de los desarrolladores sigan trabajando.
- Usar los tests “más caros” contra builds funcionales.

Si vamos a realizar pruebas automáticas es necesario planificar nuestro desarrollo de acuerdo a los tipos de pruebas que vamos a realizar. Nuestro videojuego no ha de implementar solo las mecánicas de juego sino que ha de estar desarrollado de tal modo que permita introducir pruebas automáticas.

Una buena práctica que facilita el desarrollo de pruebas automáticas es el diseño de una arquitectura que separa la capa de presentación (cómo se ve y se interactúa con el juego) de la capa de la lógica (las mecánicas en sí mismas del juego). De este modo, podemos probar nuestra lógica independientemente de si el sistema de input es el de un dispositivo controlado por un humano, scripts de desarrollo de tests o trazas de ejecución (generalmente, de sesiones de juego de humanos).

Testing	Automático	Manual
Coste preparación	ALTO	BAJO
Coste ejecución	BAJO	ALTO
Número ejecuciones	ALTO	BAJO
Características	Rígido, Frágil, Predefinido	Ágil, Adaptativo, Holístico

Figura 6.4: Comparativa entre testing manual y automático

Un ejemplo de esta arquitectura es la empleada en Sims Online (2003), que se muestra en la Figura 6.5. Una arquitectura similar se emplea en League of Legends, como veremos más adelante:

Definición de casos de prueba

Cada videojuego tiene sus propia forma de definir sus casos de prueba automáticos. Sin embargo, para la realización de pruebas funcionales y de integración hay una tendencia hacia el uso de lenguajes de scripting que permiten implementar estos casos de prueba de una manera similar a como lo hacemos con los test de unidad (Figura 6.6).

Estos lenguajes son dependientes del dominio (videojuego) y obligan a implementar funcionalidades específicas en nuestro juego (como la capacidad de crear personajes en lugares específicos del juego, teletransportar al jugador, configurar un estado concreto del juego...)

Algunas acciones programadas en el script no son inmediatas por lo que es necesario establecer cierta sincronización tras la petición de realización de una acción (comando remoto). En general existen dos tipos de sincronización:

- `WaitTime(n)`: espera n segundos antes de realizar la siguiente acción. Es dependiente de la máquina.
- `WaitUntil(cond)`: espera a que se cumpla cierta condición. Más realista y no presupone que una acción ha de realizarse en un determinado periodo de tiempo.

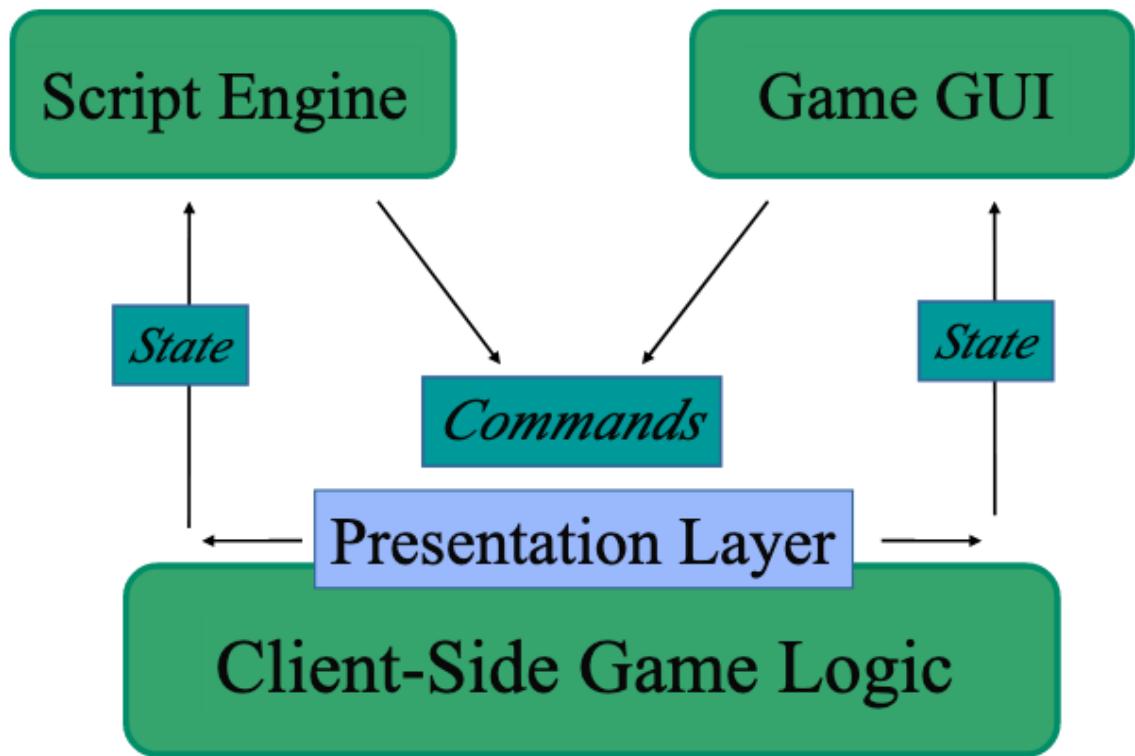


Figura 6.5: Arquitectura para testing automático. Fuente: [Automated Testing of Massively Multi-Player Games](#). Larry Mellon (2003 y GDC 2006)

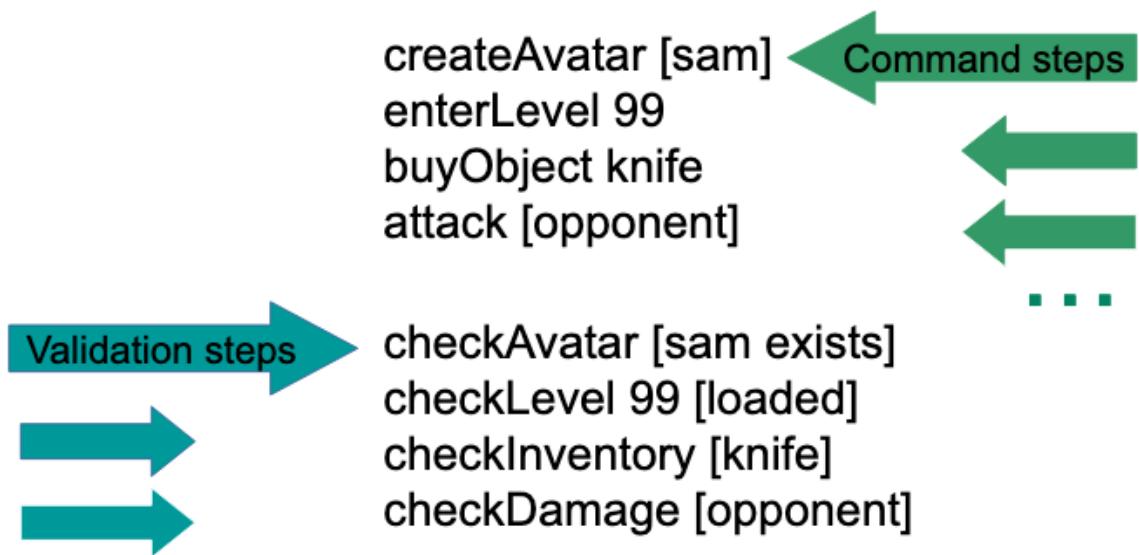


Figura 6.6: Lenguajes de scripting para pruebas

Ejemplos de uso de pruebas automáticas en la industria

‘Final Fantasy VII’ Remake - Square Enix

AutoQA es un sistema que les ayuda en la automatización de las tareas de QA, basado en una combinación de replay y exploración (Figura 6.7).

Han diseñado un sistema de replay con una gran parte genérica y reutilizable para otros videojuegos.

Los datos de replay que se almacenan son:

- Input del jugador
- Estado de juego, tanto genérico como específico

El game state genérico almacena:

- Dónde:
 - Posición, level id
 - Velocidad, orientación
- Cuándo
 - Tiempo: tiempo desde el inicio de juego, UTC, real, número de frames...

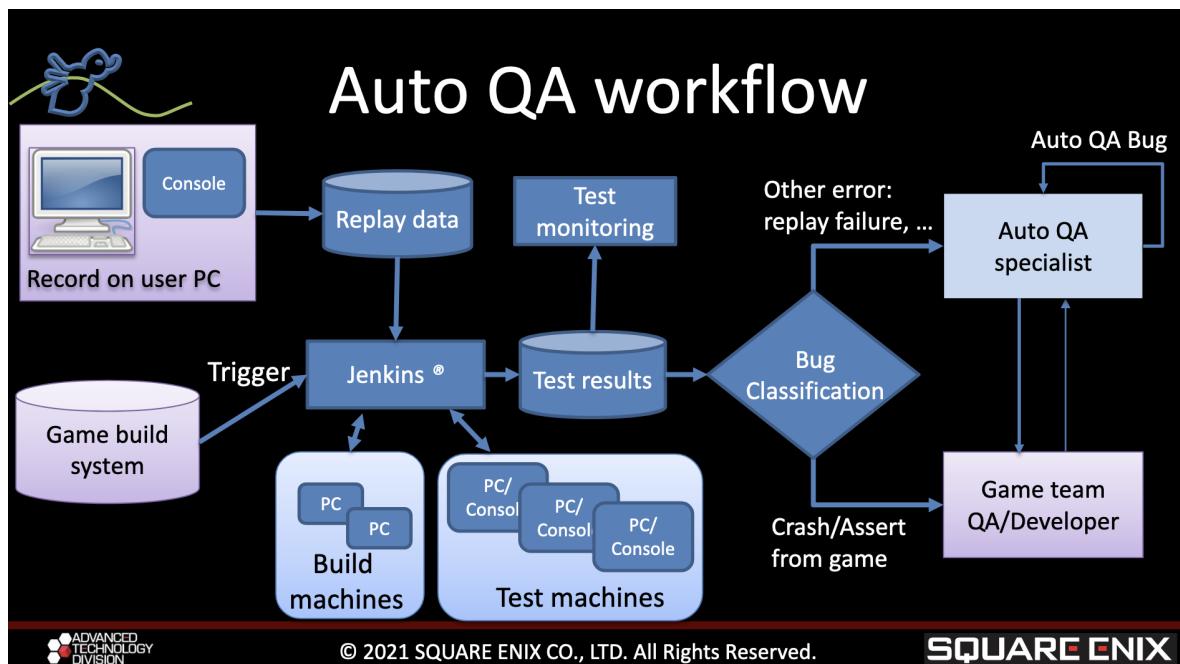


Figura 6.7: Automatización de QA en Square Enix

- Qué
 - Acciones y eventos: pulsaciones, eventos de pad, eventos de juego...
- ¿Por qué?
 - Estado de juego con información sobre el tipo de juego, el usuario, el tipo de estado y flags de sincronización. Los estados se apilan para facilitar el trabajo de algunos bots.

La exploración consiste en usar bots que construyen mapas en forma de cuadrícula en 3D dinámicamente. De esta forma, no dependen de las mallas de navegación reales que usarán los NPCs del juego, de modo que se pueden testear mejor los niveles del juego. Se usan también para detectar errores de colisiones.

Finalmente, tienen distintos bots que son capaces de, usando toda esta información, combinar replay con exploración, de modo que son capaces de testear gran parte del nivel a partir de una traza sencilla de ejecución de un jugador.

Fuentes:

- [Final Fantasy VII Remake Automating Quality Assurance and the Tools for the Future](#). Fabien Gravor (Square Enix). GDC 2022.

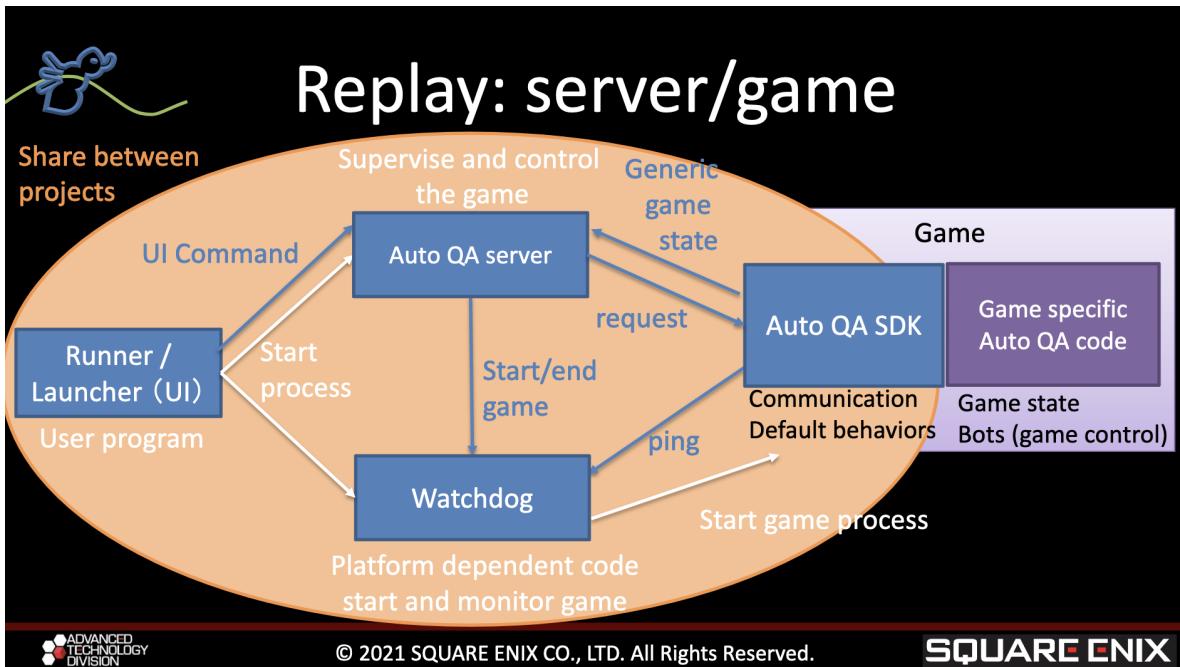


Figura 6.8: Arquitectura de AutoQA en Square Enix

Sea of Thieves - Rare

Tipos de tests que realizan:

- Tests de unidad usando NUnit y el sistema de automatización de Unreal (similar al Unity Test Runner). La duración es de unos 0.1s por test.
- Tests de integración creando escenas concretas en Unreal y programando los tests mediante blueprints. La duración es de aprox. 20s por tests (tiempo de carga de los assets, conexión de red...)
- Actor tests: Tests de integración creados desde código (no desde blueprints). Por debajo crean una escena (Arrange), ejecutan una acción (Act) y comprueban que el estado final obtenido concuerda con el esperado (Assert). Son más rápidos que los de integración por lo que usan los actor tests para casos de error y los de integración para los “golden path” (secuencia de pasos que lleva a un estado satisfactorio). Hay una relación 12:1 entre los actor tests y los test de integración.
- Auditoría de assets: comprobación de assets y datos usados en el juego.
- Screenshot: hacen capturas de pantalla en ciertos lugares y comprueban (con una captura correcta) que lo que se ve está correcto.
- Rendimiento: tiempos de carga, uso de memoria y framerate en distintos lugares

del juego.

- Bootflow: relacionados con la comunicación cliente-servidor.



Figura 6.9: Tests de integración

```
IMPLEMENT_TEST_SHADOWSKELETON( InDarkState_NowDayTime_ChangesToLightState )
{
    AShadowSkeleton& ShadowSkeleton = SpawnShadowSkeleton();
    ShadowSkeleton.SetCurrentState( EShadowState::Dark );

    SetGameWorldTime( Midday );
    ShadowSkeleton.Tick( 1.0f );

    TestEqual( ShadowSkeleton.GetCurrentState(),
               EShadowState::Light );
}
```



Figura 6.10: Actor tests

Usan TeamCity como sistema de integración continua. El flujo de trabajo usado para el sistema de control de versiones es el *Trunk-based aproach*, que facilita la integración continua y resuelve el problema de cuánto cuesta combinar e integrar los cambios de diferentes ramas. Las ramas se pueden usar pero suelen ser muy pequeñas (máximo una persona, dos días).

Tienen un proceso de “pre-commit” que hace una build de los cambios locales y ejecuta un número mínimo de pruebas antes de hacer la subida. Posteriormente, si se pasan los test anteriores, se hace la build y ejecuta todos los tests (de manera extensiva). En general se realiza cada 20 minutos. El resultado se proyecta en pantallas (Figura ??) y si ha habido un fallo se indica lo que ha fallado y de quién es la última build que ha fallado. Además, se detienen los commits (nadie puede hacer commits con la pantalla en rojo).

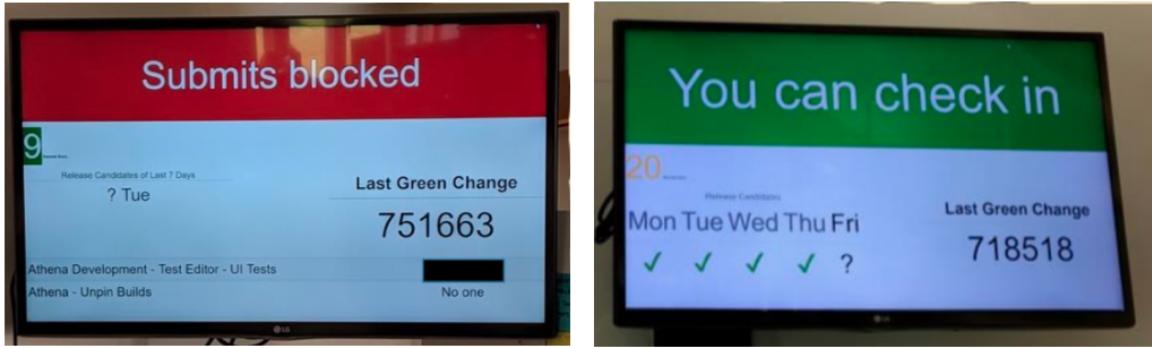


Figura 6.11: Pantalla del sistema de integración continua

Cada nueva funcionalidad implementada ha de desarrollar pruebas automáticas que la verifiquen. Lo hacen los propios desarrolladores, no hay responsables de pruebas.

Después pasan por un testeo con probadores humanos sobre una build estable. Finalmente pasan esas builds a jugadores para que den feedback del juego.

Para evitar *flaky tests*, repiten los tests que han fallado, de modo que un test que falla dos veces se considera genuino. En caso contrario, se considera intermitente.

Los tests que fallan constantemente se meten en cuarentena para comprobar si se han implementado correctamente. Estos tests no paralizan el proceso de build.

Finalmente, usan tests automáticos para los shaders de partículas (principalmente, los que implementan el comportamiento del océanos). Los problemas de estos tests es que hay que crear una infraestructura sobre la suite de tests de unidad que tiene Unreal para mover los parámetros de prueba al shader y para , una vez ejecutado en la GPU, devolverlos a la suite de tests de unidad. Los combinan con screenshot tests para verificar no solo el comportamiento sino el resultado visual de los shaders.

“Actualmente” (2019) tienen unos 23000 tests: unidad (22 %), actor (70 %) e integración (5 %) son los más abundantes. Tienen de screenshot (1,4 %), de rendimiento (0,5 %) y de bootflow (0.005 %). Adicionalmente, tienen otros 81700 tests para auditar los assets.

Beneficios y reflexiones:

- Reducen el tiempo de verificación de la build a 1,5 días (en Sport Rivals tardaban 10 días)
- Reducen el número de probadores (de 50 a 17). Implica menor coste y mejora el trabajo de este equipo al ser más pequeño.

- Se reduce el número de bugs abiertos a lo largo del tiempo (el número máximo es 214, frente a los 3000 que tuvieron abiertos en Nuts and Bolts)
- Reduce los crunch ya que hacen que el desarrollo sea sostenible.
- La velocidad de trabajo es aproximadamente la misma, ya que el tiempo que se invierte en desarrollar los tests es aproximadamente el mismo que se dedicaba en corregir bugs, con las ventajas anteriores.
- La infraestructura e implementar tests adecuados lleva tiempo. Poner en marcha este proceso lleva tiempo.
- En ocasiones hay que ser pragmáticos y tener en cuenta que cubrir todos los posibles casos (el testeo perfecto) es imposible.

Fuentes:

- Automated Testing of Gameplay Features in ‘Sea of Thieves’. Robert Masella. Rare Ltd. GDC 2019: <https://www.gdcvault.com/play/1026366/Automated-Testing-of-Gameplay-Features>
- Automated Testing at Scale in Sea of Thieves. Jessica Baker. Unreal Fest Europe 2019: <https://youtu.be/KmaGxprTUfl>
- Adopting Continuous Delivery. Jafar Soltani . Rare Ltd. GDC 2018: <https://www.youtube.com/watch?v=cKfz2nEgaX8>
- Blog de desarrollo de Rare: <https://www.rare.co.uk/news/tech-blog-testability>
- Lessons Learned in Adapting the ‘Sea of Thieves’: Automated Testing Methodology to ‘Minecraft’. GDC 2021. <https://www.linkedin.com/pulse/gdc-2021-lessons-learned-adapting-sea-thieves-testing-henry-golding/> GDC Vault: <https://www.gdcvault.com/play/1027345/Lessons-Learned-in-Adapting-the>
- Automated Testing of Shader Code. GDC 2024. <https://www.gdcvault.com/play/1034202/Automated-Testing-of-Shader>

Call of Duty - Activision

En Activision utilizan un CI implementado desde cero llamado Compass. Nace en 2011 durante el desarrollo de Destiny y se ha utilizado en el desarrollo de los Call of Duty.

Estadísticas de la infraestructura de Compass:

- 700 PCs
- 300 DevKits (Xbox y PS4)
- 900 usuarios
- Ejecuta unas 50000 tareas al día

Realiza distintos tipos de tareas, con distinta temporalización:

- Revisa cada subida al repositorio y compila el juego y las herramientas, hace las conversiones necesarias de assets, ejecuta todos los tests y varios mapas en distintas plataformas. Permite varias builds en paralelo y se realiza en menos de 30 minutos.
- Allmaps: Se ejecuta aproximadamente cada hora. Ejecuta el arranque de todos los mapas y va haciendo capturas de pantallas en lugares concretos para verificar que se está ejecutando correctamente.
- Release builds: Versiones completas del juego para QA o para publicar.
- Mantenimiento de ramas: merges automáticos de ramas.

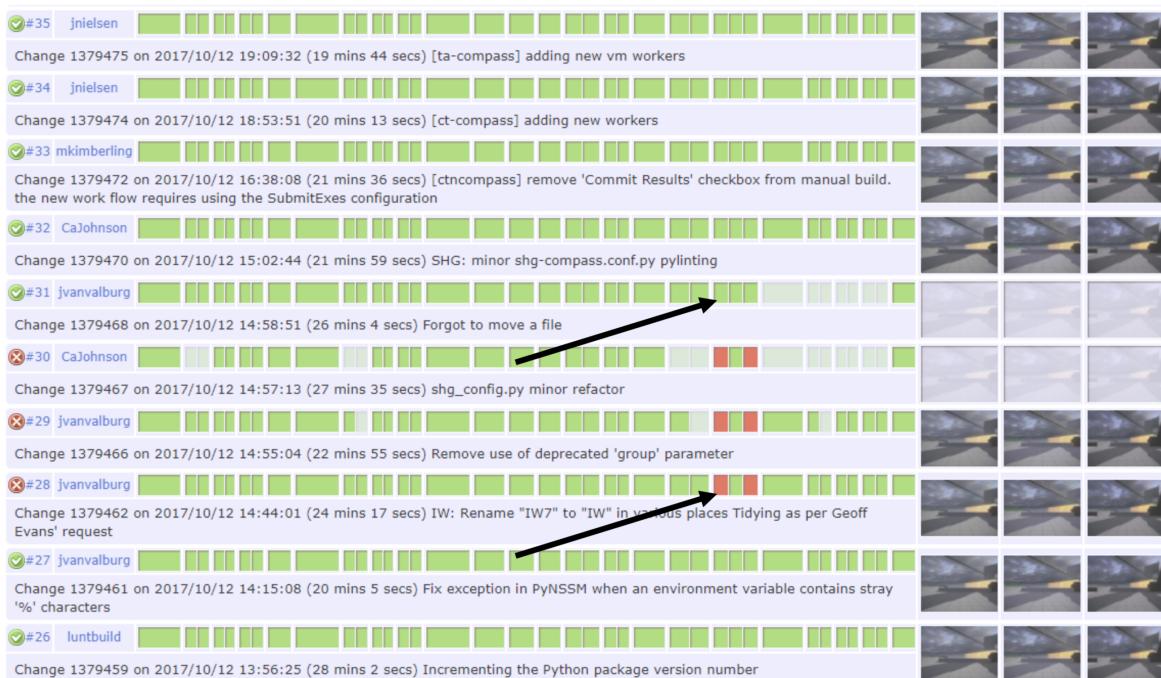


Figura 6.12: Consola de Compass

El sistema genera estadísticas de diferente índole de cada una de las pruebas realizadas. Por ejemplo, cada noche se ejecutan unas pruebas especiales (*nightly builds*):

- Carga de cada uno de los mapas.
- Teletransporte del jugador a lugares críticos del escenario (decidido por resultados del departamento de QA)
- Dejan al personaje durante 20-30 segundos.
- Capturan datos de rendimiento: FPS, uso de CPU y GPU, memoria...



Figura 6.13: Acceso a errores en Compass



Figura 6.14: Información agregada de la sala de desarrolladores (Infinity Warfare)

- Captura de pantalla para ver que aparece lo que debe de aparecer.



Figura 6.15: Estadísticas en Compass (número de shaders por versión)

Las tareas de automatización se definen mediante scripts en Python. Las principales ventajas son:

- Legibilidad.
- Depuración de pruebas en local.
- Se pueden usar otros módulos de Python.
- Se pueden tener en un sistema de control de versiones.

El principal inconveniente es la curva de aprendizaje.

El sistema también es capaz de hacer *bucketing* de errores: analiza las primeras líneas de las pilas de llamadas devueltas por el error para componer un “resumen” y un identificador del error producido y así poder hacer un seguimiento automático del mismo (cuántas veces se ha producido, desde cuándo se viene produciendo...)

Para los problemas relacionados con Flaky tests, se reintenta la ejecución de unas

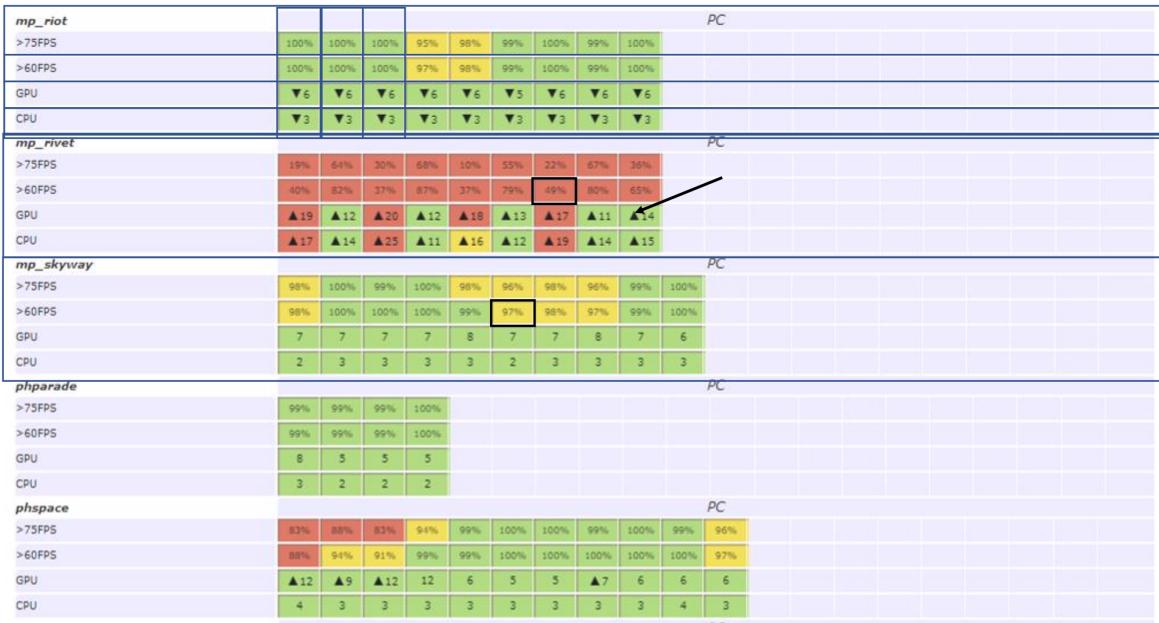


Figura 6.16: Estadísticas recogidas por las nightly builds

pruebas que han producido error, con el fin de comprobar si se puede reproducir el error. Además, permite cambiar de manera automática el recurso en el que se ejecuta, para evitar problemas relacionados con el recurso seleccionado (p.e. una consola desconectada o defectuosa)

Fuentes:

- Automated testing for Call of Duty. Jan van Valburg. Activision (Digital Dragons 2018): <https://youtu.be/6OVFbLnIFR4>
- Automated Testing and Profiling for ‘Call of Duty’. Jan van Valburg. Activision (GDC 2018): <https://www.youtube.com/watch?v=8d0wzyiikXM>

Ubisoft: The division

Generan escenarios de manera procedural. Estos escenarios son enormes, pudiendo producir hasta 100 horas de juego. Además, es un juego que está siendo actualizado constantemente.

Crean una IA que es capaz de controlar las entradas del usuario para crear bots que navegan por el mundo para la realización de ciertas pruebas. Estos bots son tratados

como jugadores humanos.

Para programarlos usan una UI o una interfaz de comandos:

```
console.cmd cmd="ClientBot.SetParam enable 1"
script.wait time=0.5

console.cmd cmd="ClientBot.SetParam destination (24.98 0.01 -21.83)"
console.cmd cmd="ClientBot.SetParam request_go_to 2"
ClientBot.WaitForAroundPosition timeout=5.0 pos="(24.98 0.01 -21.83)" radius=3.0
console.cmd cmd="ClientBot.SetParam request_go_to 1"

console.cmd cmd="ClientBot.InjectGameAction Roll"
script.wait time=1

console.cmd cmd="ClientBot.SetParam destination (36.90 0.01 -21.96)"
console.cmd cmd="ClientBot.SetParam request_go_to 2"
ClientBot.WaitForAroundPosition timeout=5.0 pos="(36.90 0.01 -21.96)"
script.wait time=0.5
console.cmd cmd="ClientBot.SetParam request_go_to 1"

console.cmd cmd="ClientBot.InjectGameAction EnterCover"
script.wait time=1

console.cmd cmd="ClientBot.SetParam destination (42.20 0.01 -25.28)"
console.cmd cmd="ClientBot.SetParam request_go_to 4"
ClientBot.WaitForAroundPosition timeout=5.0 pos="(42.20 0.01 -25.28)"
script.wait time=1
```

Figura 6.17: Consola de comandos de los bots de pruebas de The Division

Lo usan para *smoke tests*:

- Cada noche ejecutan las misiones de Manhattan (5 horas)
- Cada fin de semana testean todo el underground (7 horas)

Otros bots deambulan por el mundo y extraen métricas de juegos que posteriormente son analizadas para verificar el comportamiento del juego. Se ejecutan por las noche y durante las 48 horas del fin de semana. Algunas de las métricas que obtienen son:

- Si todos los lugares del mundo son alcanzables con las mallas de navegación y la IA (*world coverage*).
- Pruebas de rendimiento: ver si hay lugares en los que bajan los FPS.

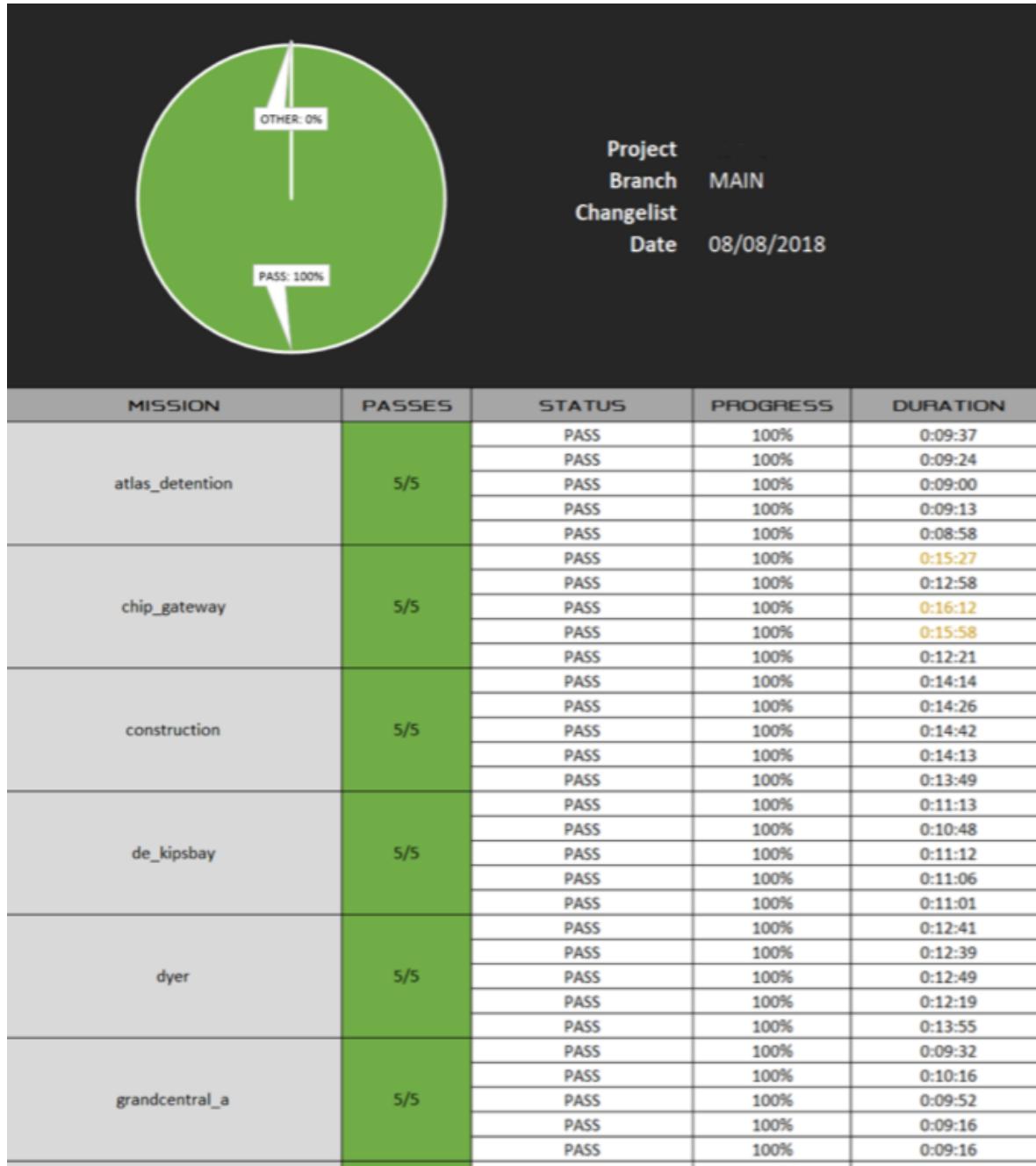


Figura 6.18: Dashboard de resultados de pruebas

- Número de mallas renderizadas.
- Áreas a las que se puede entrar pero no se puede salir (*Stuck detection*).



Figura 6.19: World Coverage

Fuentes:

- Automated Testing: Using AI Controlled Players to Test ‘The Division’. Jose Paredes, Pete Jones. Ubisoft. GDC 2019: <https://www.gdcvault.com/play/1026382/Automated-Testing-Using-AI-Controlled>

Mercury Steam: SpaceLords (a.k.a. Raiders of the broken planet)

Problemas de combinatoria: 20 personajes x 4 armas con múltiples skins, modificadores y stats para personajes y armas hacen que existan millones de combinaciones.

El sistema de automatización usa Jenkins para hacer lo siguiente:

- Build del código
- Ejecución del camino crítico básico (conexión con el servidor)
- Carga de todas las misiones para comprobar que todos los archivos necesarios están disponibles (3-4h)



FPS < 25

Figura 6.20: Rendimiento: lugares en los que los FPS bajan por debajo de un umbral

- Tests de continuidad: se puede ganar-perder en todas las misiones y se dirige al menú/nivel correctos.
- Copia en PCs (27) de QA para realizar testing automático durante la noche y pruebas con personal de QA (*testing creativo*).
- La información de las ejecuciones se guarda en trazas. Se genera un resumen con PowerBI a modo de dashboard en el que cada mañana se puede ver el resumen de las ejecuciones de la noche.

Testing automático:

- Bots para probar mallas de navegación: comprobación de que la malla completa es navegable, que no hay lugares inalcanzables, que los navlinks están posicionados correctamente...
- Múltiples IAs juegan partidas mientras un tracker va almacenando información sobre crashes. Estas IAs graban en una cola circular información de los N últimos frames de ejecución y cuando se produce un error se hace un dump de esa cola. Ese dump se puede reproducir visualmente, lo que hace que se pueda ver dónde ha ocurrido el error. Esta herramienta ha podido reducir la tasa de errores relacionadas con la IA de un 22% a un 1%.



Mesh Count

Figura 6.21: Rendimiento: número de mallas renderizadas en cada posición

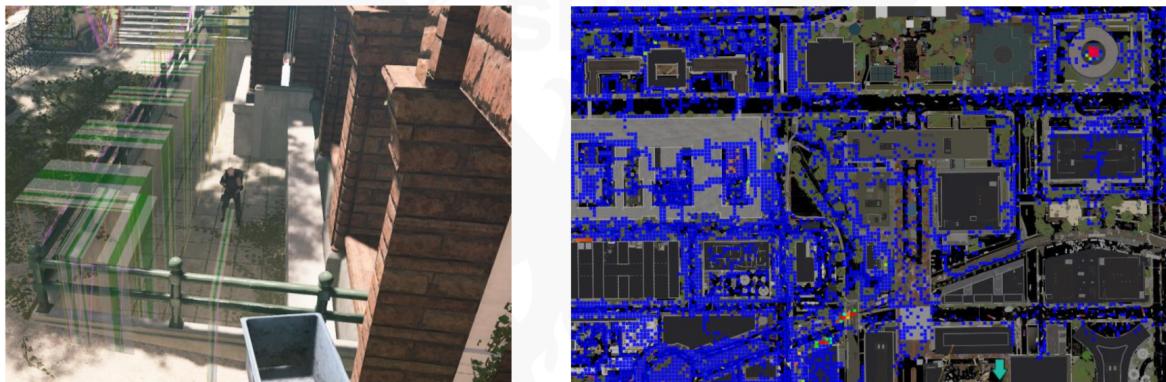


Figura 6.22: Stuck detection: áreas a las que se puede llegar pero de las que no se puede salir

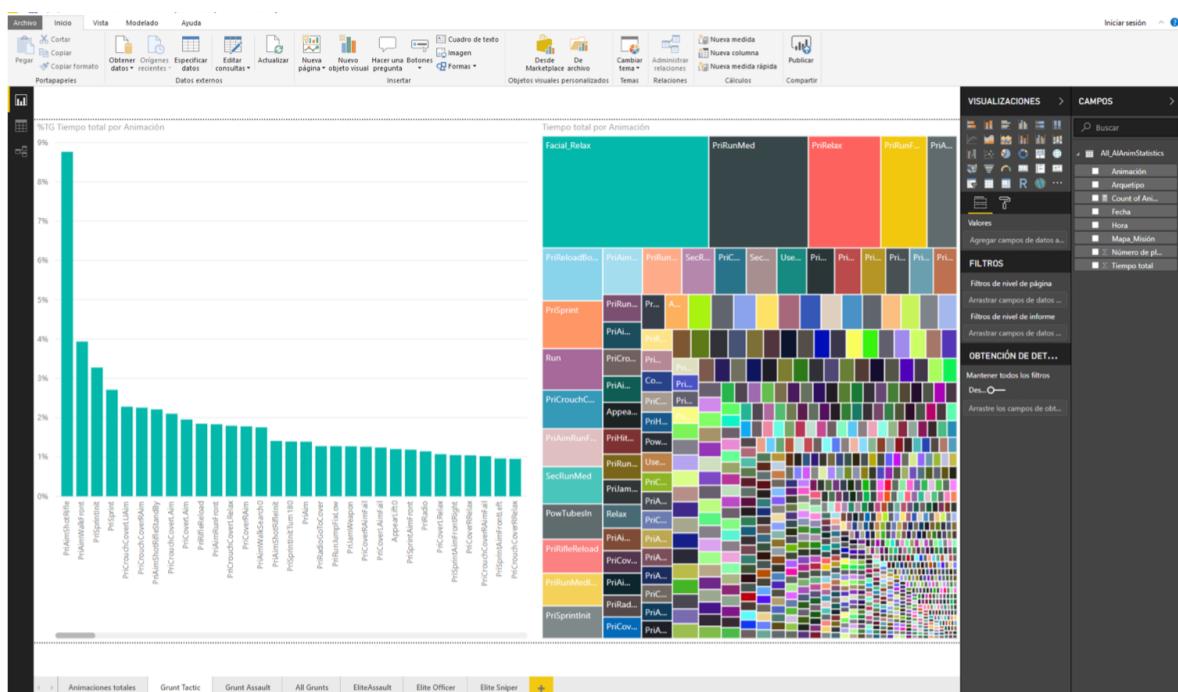


Figura 6.23: Dashboard de datos de ejecución automática con PowerBI

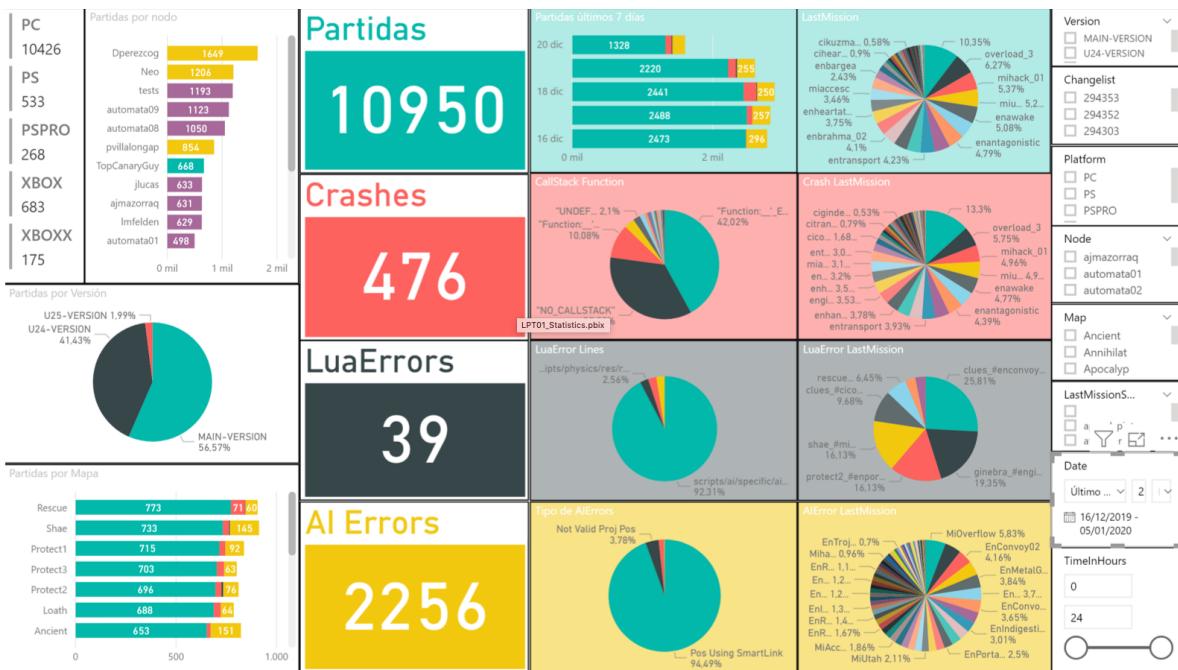


Figura 6.24: Dashboard de datos de ejecución automática con PowerBI

- Tracking de animaciones. Usan analíticas para saber cuáles son las animaciones que más se usan en el juego. La principal conclusión extraída de la prueba inicial es que el 30 % de las animaciones no se usan o que hay animaciones que se usan demasiado. Sirve para mejorar el pipeline de trabajo (producción no pide tantas animaciones a arte), para elegir qué animaciones pulir de cara a versiones finales y qué animaciones eliminar para reducir el consumo de memoria del videojuego.

League of Legends - Riot Games

Tienen equipos de DevOps, responsables de la creación de herramientas de testeo automático y que ayudan a los equipos de desarrollo a escribir mejores casos de prueba. No reemplazan el testeo manual (QA) pero sí que les ha ayudado a agilizar el bucle de desarrollo-pruebas y a liberar a los probadores del testeo más sistematizado, de modo que estos se enfocan en las pruebas destructivas.

Tienen su propio sistema de integración continua o *Build Verification System* (BVS):

- Sistema distribuido que ejecuta unos 100000 casos de prueba al día.
- Ha de gestionar 100 check-ins (commits) cada día

- Resultado de los tests se ejecuta en un máximo de 1 hora
- Prueban tanto el cliente del jugador como el servidor.
- Se encarga de:
 - Adquirir y compilar los assets y artefactos necesarios para el test.
 - Hacer una build y cargarla en una máquina de pruebas
 - Ejecutar el sistema de pruebas y los casos de prueba
 - Recopilar los datos y generar informes de resultados de las pruebas.

Los tests están implementados en Python y suelen ser similares a los tests de unidad (instrucciones de configuración, acción y consulta para verificar el cambio de estado). Algunos tests requieren de esperas condicionales ya que dependen de la velocidad de la máquina en la que se están ejecutando.

Usan RPCs para ejecutar los test y monitorizar el estado del juego, se pueden ejecutar en local y en los servidores. Los resultados (datos) de ejecución se almacenan en un servidor aparte durante 6 meses por si es necesario analizarlos.

Si un test falla en un servidor de test:

- Se crea un “bug ticket”.
- Se envía un mail a quien ha hecho el commit del error

El BVS separa el sistema de ejecución de tests de lo que es el reporte: los datos generados por el primero se envían al servicio de reporting, que lo agrega y guarda para ver la frecuencia de ciertos bugs, cuándo se han resuelto, etc.

Rendimiento de BVS:

- Ejecutan unos 5500 casos de prueba en 18 minutos.
- El tiempo medio de detección de un fallo (desde que se hace el commit hasta que se reporta) es inferior a una hora.
- El 50 % de los errores críticos son encontrados por el sistema de pruebas automático. El resto lo detecta la gente de QA o en el PBE de Riot Games, que es el entorno de pruebas en el que se puede jugar con las características nuevas o experimentales del juego antes de su lanzamiento

Fuente:

- Blog de desarrollo de Riot Games: <https://technology.riotgames.com/news/automated-testing-league-legends>
- <https://technology.riotgames.com/news/legends-runeterra-cicd-pipeline>

Candy Crush Saga - King

Candy Crush Saga - King

Desde 2019, los diseñadores de Candy Crush Saga son capaces de crear 45 niveles semanales, teniendo como objetivo alcanzar los 18000 en 2024. Esto lo consiguen gracias a dos herramientas fundamentales de soporte:

- Playtesting bots: bots que juegan automáticamente a los niveles creados por el diseñador (integrado en la herramienta de diseño de niveles) y generan métricas de dificultad, shuffles, número de movimientos... Estos bots son entrenados usando aprendizaje supervisado con las trazas de los jugadores. Estos bots tardan horas (menos de un día) en ser entrenados. Cuando hay elementos de juego nuevos entonces usan aprendizaje por refuerzo y roles de juego humanos para conseguir que los bots puedan seguir jugando a estos nuevos niveles. Estos bots tardan aproximadamente una semana en ser entrenados .
- Tweak system: en un sistema por lotes que usa algoritmos genéticos para ajustar la dificultad de un nivel. El diseñador indica los parámetros del nivel esperados (los actuales los obtiene del playtesting bot) y un algoritmo genérico hace variaciones sobre el nivel (eliminación de elementos, cambios de tipo/color...) para quedarse con los niveles que mejor se ajusten. Cada nivel es jugado por un playtesting bot para obtener valor de fitness (scoring). La función de scoring no solo tiene en cuenta lo devuelto por el bot sino también la similitud con el nivel inicial del diseñador y otros parámetros adicionales de diseño (restricciones de tipos de elementos, número de movimientos máximos...)

Gracias a estos sistemas reducen los ajustes manuales de dificultad en un 95 % y estiman que el tiempo de creación de niveles es un 50 % más rápido.

Fuente:

- Levelling Up: How AIs Transformative Role in Level Automation Production Adds Business Value in ‘Candy Crush Saga’. GDC 2024. <https://www.gdcvault.com/play/1034366/Levelling-Up-How-AI-s>

Otros recursos y charlas

Otros recursos y charlas

- Jonas Gilberg (EA):

- AI for Testing The Development of Bots that Play ‘Battlefield V’. GDC 2019.
https://www.youtube.com/watch?v=_cslewPyKks
- AI Summit AI for Testing at EA From ‘Star Wars’ to ‘Apex’ and Beyond. GDC 2023. <https://gdcvault.com/play/1028924/AI-Summit-AI-for-Testing>
 - <https://autotestingroundtable.com/>

Nota final

Esta obra está bajo una Licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional

