GAÑANADA Nº 005

# El Singleton que terminó siendo una religión

· · ·

| | |
|---|---|
| CATEGORÍA | ABSTRACCIONES INNECESARIAS |
| GRAVEDAD | ■■■■□ IV — EXTREMO |
| FECHA DEL HALLAZGO | 15 DE OCTUBRE DE 2026 |
| ETIQUETAS | #singleton #estado-global #inyeccion-de-dependencias #cpp #testing |

§ 1

## Descripción del hallazgo

Génesis, 2015. En el principio era `ConfigManager::instance()`, y su propósito era bueno: leer un archivo de configuración una sola vez.

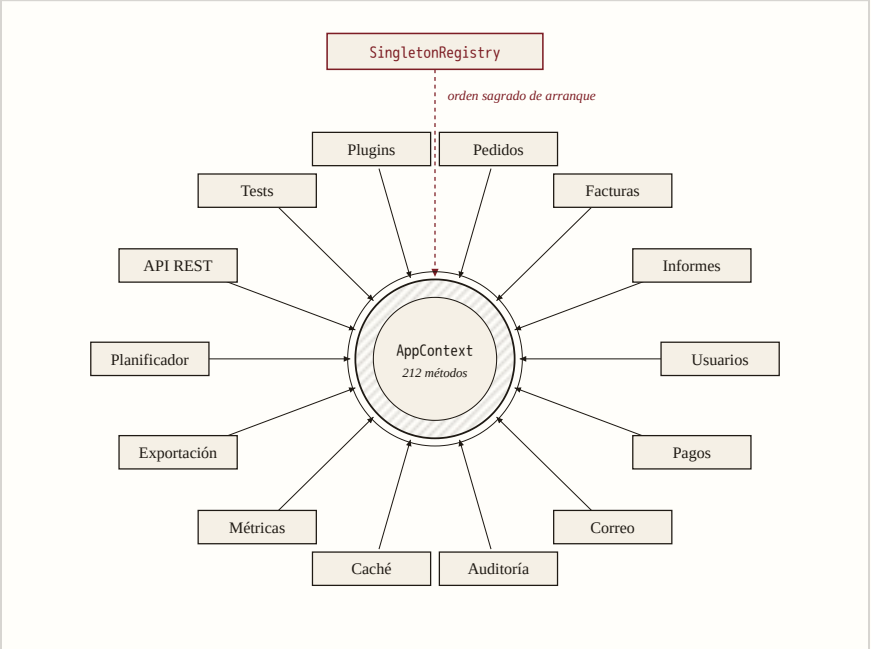
Después vino la expansión. Ya que estaba ahí y todo el mundo podía alcanzarlo, se le añadió el logger. Luego la conexión a la base de datos, el usuario actual, la caché, los *feature flags* y las métricas. En 2018 fue rebautizado como `AppContext`, nombre que reflejaba mejor su naturaleza: el contexto de todo.

Cuando alguien observó que aquello era difícil de probar, el clero respondió con más liturgia:

```
class AppContext : public Singleton<AppContext> {
    friend class SingletonRegistry;
public:
    Config&     config();
    Logger&     logger();
    Database&   db();
    User&       currentUser();
    // … 208 métodos más
    void resetForTesting();   // llamar en cada setUp(). Siempre. Sin excepciones.
};

template <typename T>
class AbstractSingletonFactory { /* fábrica de instancias únicas */ };
```

Una fábrica cuyo propósito es fabricar objetos de los que solo puede existir uno. El espécimen ha alcanzado la autoconciencia teológica.




*Lám. 5.1 — Liturgia del Singleton. Todos los caminos conducen a `AppContext`; ninguno sale de él.*

§ 2

## Contexto del descubrimiento

El arranque de la aplicación sigue un ritual que nadie entiende del todo y nadie se atreve a alterar:

```
int main(int argc, char** argv) {
    SingletonRegistry::instance().bootstrap();          // orden sagrado: no
alterar
    AppContext::instance().config().load(argv[1]);
    AppContext::instance().db().connect();              // necesita config()
    AppContext::instance().logger().setLevel(
        AppContext::instance().config().get("log.level"));
    // el logger escribe auditoría en db(): por eso va después. Creemos.
}
```

El hallazgo se produjo al actualizar el compilador. La aplicación empezó a fallar al arrancar, pero solo en una plataforma. Uno de los singletons se construía como objeto estático global, en una unidad de traducción distinta de otro del que dependía, y el estándar de C++ no garantiza el orden de inicialización dinámica entre unidades de traducción. Durante años había funcionado por el orden en que el enlazador recibía los archivos objeto. El nuevo enlazador tenía otras creencias.

§ 3

## Consecuencias observadas

- **Dependencias ocultas.** Cualquier función puede usar cualquier cosa sin declararlo; las firmas dejan de describir lo que el código necesita.
- **Tests acoplados entre sí.** El estado compartido hace que los tests dependan del orden de ejecución y fallen al paralelizarse. `resetForTesting()` reduce el problema; olvidarlo una vez lo multiplica.

- **Imposible tener dos configuraciones en el mismo proceso**: ni dos clientes, ni dos entornos, ni un test que compare el antes y el después.
- **Recompilación masiva.** `AppContext.h` está incluido en más de seiscientos archivos; cualquier cambio en él recompila el proyecto entero.
- **El fiasco del orden de inicialización estática**, que convierte una actualización del compilador en un incidente.

§ 4

## Recomendación de la academia

Lo que el sistema necesitaba era *una sola instancia*, no *acceso global* a ella. Son cosas distintas, y la primera no exige la segunda: basta con crear cada objeto una vez, en un único lugar, y pasárselo a quien lo necesite.

```
int main(int argc, char** argv) {
    const Config config = Config::load(argv[1]);
    Logger       logger{config.logLevel()};
    Database     db{config.database(), logger};
    OrderService orders{db, logger};
    return Server{config.port(), orders, logger}.run();
}
```

El orden de construcción es explícito y lo comprueba el compilador: no se puede construir `Database` sin un `Logger`. Cada clase declara sus dependencias en el constructor, y los tests le pasan las suyas sin rituales de limpieza.

Conviene no sustituir una religión por otra. No hacen falta un contenedor de inyección de dependencias ni una interfaz por cada clase: una raíz de composición escrita a mano en `main` suele ser suficiente, y las interfaces solo merecen existir donde hay una sustitución real —un reloj, una red, un dispositivo—.

*No trabajes así, no seas un gañán.*

## REFERENCIAS

1 ISO C++ FAQ — «What's the static initialization order fiasco?»

2 Mark Seemann, «Dependency Injection Principles, Practices, and Patterns» — *El concepto de raíz de composición («composition root»).*

Separata del volumen I de la Bitácora del Gañán, publicada por AXXA.
https://axxa.embeddedsource.es/bitacora-del-ganan/005-el-singleton-que-termino-siendo-una-religion