Batiburrillo del programador

Cosas que me sirven y que sirven

- Creación de diagramas DBML
- Borrar archivos o directorios que comienzan por letra (A–Z/a–z)
- Guía de Semantic Versioning y Git Flow
- Análisis: Principios de Good Code en Laravel

Creación de diagramas DBML

Introduccción

Me gusta tener un cuadro de lo que tengo en Mysql. A los largo de los años, ha dio cambiando el formato, las utilidades. Lejos queda el insoportable Workbench de mysql.

Ahora puedo usar TablePlus, pero a la postre muchas veces eso se me queda corto para lo que me gusta.

Asi que ahor aprefioer mantener la logica guardada en formato DBML, lo cual me permite una visión más adecuada, y real de lo que hay y lo que no hay en la app, y que uso a menudo con DBDiagram

db2dbml

db2dbml es una puntenteherramienta para generar un fichero DBML contectando a un servidor o motores de basos de datos.

Un ejemplo abajo de como generar en local el diagrama de mi proyecto.

```
❯ dbdocs db2dbml mysql
'mysql://root:MYPASSWORD@localhost:3306/lowino?socketPath=/tmp/mysql.sock' -o
notas/DBML/lowino.dbml
✔ Connecting to database... done.
✔ Generating DBML... done
✔ Wrote to notas/DBML/lowino.dbml
```

Tras eso tenemos el codigo en el fichero

```
Table "advert_properties" {
  "id" "bigint unsigned" [pk, not null, increment]
  "advert_id" binary(26) [not null]
  "property_id" "bigint unsigned" [not null]
  "value" text
  "created_at" timestamp
  "updated_at" timestamp
```

```
}

Table "adverts" {
  "id" binary(26) [unique, not null]
  "user_id" binary(26) [not null]
  "category_code" "mediumint unsigned" [not null, default: 1, note: 'without classification']
  "title" varchar(80) [not null]
  "slug" varchar(80) [unique, not null, note: 'Integrate -YYMMDDNN']
  "advert" text [not null]
  "price" "bigint unsigned"
  "currency_id" "tinyint unsigned" [not null, default: 1]
  "details" json
  "published_at" timestamp
  "expire_at" timestamp
  "created_at" timestamp
  "updated_at" timestamp
  "deleted_at" timestamp

  Indexes {
    published_at [type: btree, name: "adverts_published_at_index"]
  }
}
...
```

Y con ello ya podemos hacwer y deshacer como queramos, para crear nuestro digrama.

> Cierto que tambien TablePlus es casi más eficaz en términos visuales, pero el dbml sirve para nuchas más cosas.

Aviso

Borrar archivos o directorios que comienzan por letra (A–Z/a–z)

Objetivo

Eliminar **todos los archivos o carpetas cuyo nombre comience por una letra**, sin distinguir mayúsculas/minúsculas.
Ideal para limpiezas de entorno sin afectar archivos ocultos ni aquellos que comiencen por números o símbolos.

?? En macOS (Zsh)

Zsh no usa `shopt`, pero tiene su propia sintaxis de globbing avanzada.

1. Ver qué se va a borrar (modo seguro)

```
setopt extended_glob
ls -d (#i)[[:alpha:]]*
```

2. Eliminar los archivos y carpetas

```
setopt extended_glob
rm -Rf (#i)[[:alpha:]]*
```

□□ Explicación técnica

- *setopt* extended_glob activa los patrones avanzados de Zsh.
- *(#i)* activa insensibilidad a mayúsculas para todo el patrón.
- *[[:alpha:]]** coincide con cualquier nombre que empiece por letra (A–Z o a–z), de forma portable.

> □□ Ojo: patrones como (#i)[a-z]* no siempre funcionan como se espera, porque [a-z] sigue limitado al rango ASCII explícito. Usar **[[:alpha:]]** es más robusto.

? En Linux (Bash)

Bash usa `shopt` para habilitar coincidencias insensibles a mayúsculas.

1. Ver qué se va a borrar

```
shopt -s nocaseglob
ls -d [a-zA-Z]*
shopt -u nocaseglob
```

2. Eliminar los archivos y carpetas

```
shopt -s nocaseglob
rm -Rf [a-zA-Z]*
shopt -u nocaseglob
```

? Explicación técnica

- *nocaseglob* hace que Bash trate los patrones como insensibles a mayúsculas.
- *[a-zA-Z]** coincide con nombres que comienzan por cualquier letra.

> ⚠ Advertencias No borra archivos ocultos (*.env, .git,* etc.) ni nombres que empiecen con números (2023-img.png).
>
> Usa ls antes de rm -Rf para confirmar qué se eliminará.
>
> Si necesitas excluir ciertos nombres, puedes añadir filtros con grep -v, find, o usar listas de exclusión.

? Alternativas útiles

Mover en lugar de borrar:

```
mkdir -p backup_letters
mv [a-zA-Z]* backup_letters/
```

Ver el tamaño de los elementos que serán eliminados:

```
du -sh [a-zA-Z]*
```

Aviso

Esta documentación y su contenido, no implica que funcione en tu caso o determinados casos. También implica que tienes conocimientos sobre lo que trata, y que en cualquier caso tienes copias de seguridad. El contenido el contenido se entrega, tal y como está, sin que ello implique ningún obligación ni responsabilidad por parte de Castris

Si necesitas soporte profesional puedes contratar con Castris soporte profesional.

Guía de Semantic Versioning y Git Flow

Semantic Versioning (SemVer)

Formato: `MAJOR.MINOR.PATCH` (ejemplo: `1.2.3`)

¿Cuándo incrementar cada número?

MAJOR (1.x.x ? 2.x.x)

Cambios que ROMPEN compatibilidad hacia atrás (Breaking Changes)

Ejemplos:

- Cambiar la firma de un método público
- Eliminar un método o clase pública
- Cambiar el comportamiento de un método de forma incompatible
- Renombrar propiedades públicas

```
// v1.x.x
public function base100Attributes(): array


// v2.0.0 - BREAKING CHANGE
public function base100Attributes(): Collection  // Cambió el return type
```

Impacto: Los usuarios DEBEN revisar su código antes de actualizar.

MINOR (x.1.x ? x.2.x)

Nuevas características que SÍ son compatibles hacia atrás

Ejemplos:

- Añadir un nuevo método público
- Añadir un nuevo trait
- Añadir parámetros opcionales a métodos existentes

- Añadir nuevas opciones de configuración

```
// v1.1.0
class Base100 implements CastsAttributes
{
    // Métodos existentes...

    // NUEVO método añadido
    public function withPrecision(int $decimals): self
    {
        // ...
    }
}
```

Impacto: Los usuarios pueden actualizar sin cambios en su código.

PATCH (x.x.1 ? x.x.2)

Correcciones de bugs que NO cambian funcionalidad

Ejemplos:

- Corregir un bug
- Mejorar rendimiento sin cambiar API
- Actualizar documentación
- Refactoring interno sin cambios en API pública
- Corregir tests

```
// v1.1.0 - Bug: redondeo incorrecto
return (int) $value * 100;  // Error

// v1.1.1 - Patch: corregir redondeo
return (int) round($value * 100);  // Correcto
```

Impacto: Actualización segura, solo mejoras y correcciones.

Estrategia de Branches

Branches Principales

1. `main` (Branch Principal)

- **Propósito**: Código estable y listo para producción
- **Protección**: Protected (no push directo)
- **Contiene**: Solo código que ha pasado tests y revisión
- **Tags**: Todas las releases se tagean desde aquí

2. `develop` (Branch de Desarrollo) - OPCIONAL

- **Propósito**: Integración de features antes de release
- **Uso**: Si tienes múltiples features en paralelo
- **Para este proyecto**: NO necesario (proyecto pequeño)

Branches de Trabajo

Feature Branches: `feature/*`

Para nuevas características

```
feature/add-precision-option
feature/base1000-support
feature/custom-rounding
```

Ejemplo de uso:

```
# Crear feature branch desde main
git checkout main
git pull origin main
git checkout -b feature/add-precision-option

# Trabajar en la feature...
git add .
git commit -m "feat: add precision option to Base100 cast"

# Cuando esté listo
git push origin feature/add-precision-option
# Crear Pull Request en GitHub
```

Bugfix Branches: `fix/*`

Para correcciones de bugs

```
fix/rounding-precision
fix/null-handling
fix/trait-initialization
```

Ejemplo de uso:

```
# Crear bugfix branch
git checkout main
git checkout -b fix/rounding-precision

# Corregir el bug
git add .
git commit -m "fix: correct rounding precision in Base100 cast"

# Push y PR
git push origin fix/rounding-precision
```

Hotfix Branches: `hotfix/*`

Para bugs CRÍTICOS en producción

```
hotfix/security-vulnerability
hotfix/data-corruption
```

Diferencia con `fix/*`:

- Hotfix: Bug crítico que necesita release inmediata
- Fix: Bug normal que puede esperar al próximo release

Release Branches: `release/*` - OPCIONAL

Para preparar una release

```
release/1.2.0
release/2.0.0
```

Uso: Solo si necesitas "congelar" features antes de release.

Workflow de Desarrollo

Opción 1: GitHub Flow (RECOMENDADO para este proyecto)

```
main (protegido)
  ↑
  └── feature/nueva-caracteristica (trabajo aquí)
  └── fix/bug-menor (trabajo aquí)
```

Ventajas:

- Simple y directo
- Ideal para proyectos pequeños-medianos
- Deployments/releases frecuentes

Flujo:

1. Crear branch desde `main`
2. Desarrollar la feature/fix
3. Abrir Pull Request
4. Code review + Tests automáticos (GitHub Actions)
5. Merge a `main`
6. Tag y release

Opción 2: Git Flow (Para proyectos grandes)

```
main (producción)
  ↑
develop (desarrollo)
  ↑
```

```
  ├── feature/feature-1
  ├── feature/feature-2
  └── release/1.2.0
```

Ventajas:

- Mejor para equipos grandes
- Releases planificadas
- Múltiples versiones en paralelo

Desventaja:

- Más complejo
- Overkill para proyectos pequeños

Proceso de Release

Release MINOR (nueva feature - 1.1.0 ? 1.2.0)

```
# 1. Asegurarte que main está actualizado
git checkout main
git pull origin main

# 2. Actualizar CHANGELOG.md
## 1.2.0 - 2025-10-15

### Added

- New `withPrecision()` method for custom decimal precision
- Support for negative values in HasBase100 trait

### Fixed

- Rounding precision issue in edge cases

# 3. Commit el changelog
git add CHANGELOG.md
git commit -m "docs: update changelog for v1.2.0"
```

```
# 4. Crear tag
git tag -a v1.2.0 -m "Release v1.2.0 - Add precision support"

# 5. Push tag y código
git push origin main
git push origin v1.2.0

# 6. Crear GitHub Release
gh release create v1.2.0 \
  --title "v1.2.0 - Precision Support" \
  --notes "See CHANGELOG.md for details"
```

Release PATCH (bugfix - 1.1.0 ? 1.1.1)

```
# 1. Checkout main
git checkout main
git pull origin main

# 2. Actualizar CHANGELOG.md
## 1.1.1 - 2025-10-12

### Fixed

- Correct rounding precision when handling values > 1000

# 3. Commit y tag
git add CHANGELOG.md
git commit -m "docs: update changelog for v1.1.1"
git tag -a v1.1.1 -m "Release v1.1.1 - Fix rounding precision"

# 4. Push
git push origin main
git push origin v1.1.1

# 5. Release
gh release create v1.1.1 \
  --title "v1.1.1 - Bugfix Release" \
```

```
  --notes "Fix rounding precision issue"
```

Release MAJOR (breaking changes - 1.x.x ? 2.0.0)

```
# 1. Crear rama para v2
git checkout main
git checkout -b release/2.0.0

# 2. Hacer cambios breaking
# ... código ...

# 3. Actualizar CHANGELOG.md con sección BREAKING CHANGES
```

Ejemplo de CHANGELOG para v2.0.0:

````
## 2.0.0 - 2025-11-01

### BREAKING CHANGES

- `base100Attributes()` now returns `Collection` instead of `array`
- Minimum PHP version raised to 8.4
- Removed deprecated `base100()` method

### Migration Guide

**Before (v1.x):**
```php
protected function base100Attributes(): array
{
    return ['price', 'cost'];
}
````

After (v2.0):
````

```
protected function base100Attributes(): Collection
{
    return collect(['price', 'cost']);
}
```

Added

- New `Base100Collection` class
- Support for custom transformers

````
Continuación del proceso:

```bash
# 4. Mergear a main
git checkout main
git merge release/2.0.0

# 5. Tag y release
git tag -a v2.0.0 -m "Release v2.0.0 - Major overhaul"
git push origin main
git push origin v2.0.0

# 6. GitHub Release con ADVERTENCIA
gh release create v2.0.0 \
  --title "v2.0.0 - BREAKING CHANGES" \
  --notes "See CHANGELOG.md for migration guide"
````

Ejemplos Prácticos

Ejemplo 1: Añadir nueva feature (MINOR)

Escenario: Quieres añadir soporte para Base1000

```
# 1. Crear feature branch
git checkout main
```
```

```
git checkout -b feature/base1000-support

# 2. Desarrollar la feature
# - Crear src/Casts/Base1000.php
# - Añadir tests
# - Actualizar README

# 3. Commits durante desarrollo
git add .
git commit -m "feat: add Base1000 cast class"
git add .
git commit -m "test: add Base1000 tests"
git add .
git commit -m "docs: document Base1000 usage"

# 4. Push y crear PR
git push origin feature/base1000-support
gh pr create --title "Add Base1000 support" --body "Adds support for base-1000 conversions"

# 5. Después de aprobación y merge
git checkout main
git pull origin main

# 6. Release como 1.2.0 (MINOR - nueva feature)
git tag -a v1.2.0 -m "Release v1.2.0 - Add Base1000 support"
git push origin v1.2.0
gh release create v1.2.0
```

Ejemplo 2: Corregir bug (PATCH)

Escenario: Hay un bug en el redondeo

```
# 1. Crear fix branch
git checkout main
git checkout -b fix/rounding-issue

# 2. Corregir el bug
# - Editar src/Casts/Base100.php
```

```
# - Añadir test que reproduce el bug
# - Verificar que el test pasa

# 3. Commit
git add .
git commit -m "fix: correct rounding for values > 10000"

# 4. Push y PR
git push origin fix/rounding-issue
gh pr create --title "Fix rounding issue" --body "Fixes #42"

# 5. Después del merge
git checkout main
git pull origin main

# 6. Release como 1.1.1 (PATCH - bugfix)
git tag -a v1.1.1 -m "Release v1.1.1 - Fix rounding issue"
git push origin v1.1.1
gh release create v1.1.1
```

Ejemplo 3: Hotfix crítico (PATCH urgente)

Escenario: Descubriste un bug que causa pérdida de datos

```
# 1. Crear hotfix branch DESDE main
git checkout main
git checkout -b hotfix/data-loss-prevention

# 2. Corregir RÁPIDAMENTE
# - Solo el fix necesario, nada más
# - Test mínimo que demuestre el fix

# 3. Commit
git add .
git commit -m "fix: prevent data loss in null handling (critical)"

# 4. Merge DIRECTO a main (sin PR si es muy urgente)
git checkout main
```

```
git merge hotfix/data-loss-prevention

# 5. Release INMEDIATA
git tag -a v1.1.2 -m "Release v1.1.2 - Critical hotfix"
git push origin main
git push origin v1.1.2
gh release create v1.1.2 --title "v1.1.2 - Critical Hotfix"

# 6. Notificar usuarios en GitHub/Packagist
```

Comandos Git Útiles

Gestión de Branches

```
# Ver todas las branches
git branch -a

# Eliminar branch local
git branch -d feature/mi-feature

# Eliminar branch remoto
git push origin --delete feature/mi-feature

# Actualizar main desde remoto
git checkout main && git pull origin main

# Crear branch desde un commit específico
git checkout -b fix/bug abc1234
```

Gestión de Tags

```
# Listar todos los tags
git tag
```

```
# Ver detalles de un tag
git show v1.0.0

# Eliminar tag local
git tag -d v1.0.0

# Eliminar tag remoto
git push origin --delete v1.0.0

# Crear tag desde un commit antiguo
git tag -a v1.0.1 abc1234 -m "Release v1.0.1"
```

Revertir Cambios

```
# Revertir un commit (crea nuevo commit)
git revert abc1234

# Revertir último commit (antes de push)
git reset --soft HEAD~1

# Descartar cambios locales
git checkout -- archivo.php
```

Checklist de Release

Antes de Release

- Todos los tests pasan (`composer test`)
- PHPStan sin errores (`composer phpstan`)
- Código formateado (`composer format`)
- CHANGELOG.md actualizado
- README.md actualizado (si hay cambios en uso)
- Versión en composer.json coincide? (NO - Packagist lo maneja)
- Pull Request revisado y aprobado

Durante Release

- Main actualizado (`git pull origin main`)
- Tag creado con mensaje descriptivo
- Tag pusheado a GitHub
- GitHub Release creada con notas
- Packagist se actualizó automáticamente (webhook)

Después de Release

- Verificar que aparece en Packagist
- Badges del README actualizados
- Anunciar en redes/comunidad (si es relevante)
- Crear issues/milestones para próxima versión

Recomendaciones Específicas

Estrategia Recomendada

Para proyecto pequeño:

1. Usar GitHub Flow (simple)
2. Main siempre deployable
3. Feature branches para TODO
4. Pull Requests siempre (aunque seas solo tú - para CI)
5. Tags para cada release

Naming Conventions

```
# Features
feature/add-base1000
feature/custom-precision
feature/collection-support

# Fixes
fix/rounding-precision
fix/null-handling
fix/trait-initialization
```

```
# Hotfixes
hotfix/security-vulnerability
hotfix/data-corruption

# Docs
docs/update-readme
docs/add-examples
docs/api-documentation

# Chores
chore/update-dependencies
chore/ci-improvements
```

Commits Convencionales

```
feat: add new feature
fix: bug correction
docs: documentation only
style: formatting, no code change
refactor: code restructure
test: add/update tests
chore: maintenance tasks
perf: performance improvements
ci: CI/CD changes

# Ejemplos:
git commit -m "feat: add withPrecision() method"
git commit -m "fix: correct rounding for negative values"
git commit -m "docs: add usage examples to README"
```

Recursos Adicionales

- **Semantic Versioning**: https://semver.org/
- **Git Flow**: https://nvie.com/posts/a-successful-git-branching-model/
- **GitHub Flow**: https://guides.github.com/introduction/flow/

- **Conventional Commits**: https://www.conventionalcommits.org/

Preguntas Frecuentes

¿Cuándo hago MAJOR vs MINOR?

MAJOR (2.0.0): Si un usuario actualiza y su código se ROMPE → MAJOR

MINOR (1.1.0): Si un usuario actualiza y todo sigue funcionando → MINOR

¿Debo crear branch para cada pequeño cambio?

SÍ. Siempre trabaja en branches, incluso para cambios pequeños:

- Permite que GitHub Actions verifique antes de merge
- Historial más limpio
- Puedes descartar fácilmente si algo sale mal

¿Cuándo hacer release?

Flexible, pero algunas guías:

- PATCH: Cuando tengas 1+ bugfix importante
- MINOR: Cuando completes 1+ nueva feature
- MAJOR: Cuando hagas breaking changes (con cuidado)

Frecuencia recomendada:

- Patches: Cada 1-2 semanas
- Minor: Cada 1-2 meses
- Major: Solo cuando sea necesario

¿Puedo cambiar un tag después de crearlo?

NO recomendado una vez pusheado. Si lo haces:

- Los usuarios que ya instalaron la versión tendrán problemas
- Packagist se confunde
- Rompe la confianza

Si DEBES hacerlo:

```
# Eliminar tag
git tag -d v1.0.0
git push origin --delete v1.0.0

# Crear nuevo tag
git tag -a v1.0.0 nuevo_commit -m "..."
git push origin v1.0.0
```

Última actualización: 2025-10-10
Autor: Abdelkarim Mateos Sanchez

Aviso

Análisis: Principios de Good Code en Laravel

Contexto

Se plantean seis reglas fundamentales basadas en el Laravel Starter Kit de Nuno Maduro. El desafío no es solo aplicarlas en proyectos nuevos, sino integrar esta filosofía en proyectos existentes.

Cambio de paradigma fundamental

Tesis central: La flexibilidad de Laravel no debe confundirse con libertad para escribir código descuidado. La strictness no es un obstáculo, es un acelerador.

Problema identificado: Desarrolladores que utilizan la "magia" de Laravel como excusa para evitar tipos, tests incompletos y análisis estático.

Análisis de los seis principios

Principio 1: Cobertura de tipos al 100%

Qué implica:

```
pest --type-coverage --min=100
```

Análisis crítico:

Este principio fuerza la intencionalidad en cada variable y parámetro. La cobertura de tipos no es documentación, es contrato.

Ventajas:

- Reduce bugs de integración en un 60-70% según estudios empíricos
- El IDE se convierte en aliado activo
- Refactorizaciones seguras sin miedo

Desventajas:

- Requiere tiempo inicial significativo en proyectos legacy
- Puede generar fricción con equipos acostumbrados a PHP dinámico
- Arrays asociativos complejos necesitan DTOs o Value Objects

Implementación gradual en proyectos existentes:

1. Activar cobertura de tipos sin bloquear CI inicialmente
2. Establecer baseline actual
3. Regla: nuevo código debe tener 100% de cobertura
4. Refactorizar módulos críticos primero
5. Incrementar baseline mensualmente

Relación con SOLID:

Dependency Inversion Principle se beneficia directamente. Interfaces tipadas fuerzan contratos claros.

Principio 2: PHPStan en nivel MAX

Qué implica:

Análisis estático sin concesiones. PHPStan nivel 9 (MAX) detecta:

- Dead code
- Posibles nulls no manejados
- Tipos incompatibles en operaciones
- Propiedades no inicializadas

Análisis crítico:

El nivel MAX expone deuda técnica oculta. No es ego, es disciplina.

Ventajas:

- Detecta bugs antes de testing
- Complementa cobertura de tipos
- Fuerza a pensar en edge cases

Desventajas:

- Output inicial puede ser abrumador en proyectos legacy
- Requiere aprendizaje de sus reglas
- Algunas reglas pueden requerir baseline temporal

Implementación gradual:

```
# Paso 1: Generar baseline
vendor/bin/phpstan analyse --generate-baseline

# Paso 2: Subir nivel progresivamente
# phpstan.neon
parameters:
    level: 5  # Empezar aquí, no en MAX

# Paso 3: Incrementar cada sprint
# Sprint 1: nivel 5
# Sprint 2: nivel 6
# Sprint N: nivel 9 (MAX)
```

Relación con Testing:

PHPStan reduce la necesidad de ciertos tests unitarios. Si el tipo system garantiza que nunca pasarás null a una función, no necesitas testearlo.

Principio 3: Cobertura de tests al 100%

Qué implica:

```
pest --parallel --coverage --exactly=100.0
```

Análisis crítico:

La palabra clave es "exactly". No 99.8%. Exactamente 100%.

Ventajas:

- Elimina código muerto rápidamente
- Tests paralelos aceleran feedback loop
- Confianza absoluta en refactorizaciones

Desventajas:

- 100% coverage no garantiza calidad de tests
- Puede incentivar tests superficiales solo por métricas
- Requiere infraestructura para tests paralelos confiables

La trampa del 100%:

Cobertura de líneas no es cobertura de comportamiento. Un test puede ejecutar línea sin validar su lógica.

Mejor enfoque:

Combinar cobertura de líneas con mutation testing:

```
# Infection PHP
vendor/bin/infection --min-msi=80
```

Implementación gradual:

1. Identificar módulos críticos de negocio
2. Llevar estos a 100% primero
3. Expandir a módulos de soporte
4. Mantener 100% en código nuevo desde día 1

Relación con SOLID:

Single Responsibility Principle facilita testing. Clases con una responsabilidad son más fáciles de testear al 100%.

Principio 4: Formateo estricto automático

Qué implica:

Laravel Pint + Prettier eliminan decisiones de estilo.

Análisis crítico:

Este es el principio más subestimado. No es cosmético, es cognitivo.

Ventajas:

- Cero tiempo en code reviews discutiendo formato
- Diffs limpios enfocados en lógica
- Onboarding más rápido

Desventajas:

- Puede generar diffs masivos en primera aplicación
- Requiere consenso del equipo
- Algunos desarrolladores lo perciben como pérdida de "estilo personal"

Implementación inmediata:

```
# Aplicar una sola vez
./vendor/bin/pint

# Pre-commit hook
# .git/hooks/pre-commit
./vendor/bin/pint --dirty
```

Este principio no tiene desventajas técnicas reales. Solo resistencia humana.

Principio 5: Control del entorno en tests

Qué implica:

- Congelar tiempo
- Fake HTTP calls
- Forzar HTTPS en tests

Análisis crítico:

Tests deben ser deterministas. El mundo exterior es no determinista.

Ventajas en Laravel:

```
// Determinismo temporal
Carbon::setTestNow('2025-01-15 10:00:00');

// Aislamiento de HTTP
Http::fake([
    'api.external.com/*' => Http::response(['data' => 'fake'], 200)
]);
```

Desventajas:

- Requiere disciplina para identificar qué fakear
- Tests de integración reales siguen siendo necesarios
- Puede ocultar problemas de configuración real

Implementación:

Separar claramente:

- Unit tests: todo fakeado
- Integration tests: servicios reales en ambiente controlado
- E2E tests: flujo completo

Relación con Testing:

Este principio hace posible el 100% de cobertura. Sin él, tests son frágiles y lentos.

Principio 6: CI como senior engineer más estricto

Qué implica:

GitHub Actions ejecutando:

- Pint (formato)
- Rector (refactoring automático)
- PHPStan (análisis estático)
- Pest (tests)
- Lint (sintaxis)

Análisis crítico:

CI no es opcional. Es el guardián de calidad 24/7.

Ventajas:

- Feedback inmediato en PRs
- Imposible mergear código que rompe estándares
- Documentación viva de estándares del equipo

Desventajas:

- Requiere configuración inicial
- Puede ralentizar merges si CI es lento
- Falsos positivos pueden frustrar

Implementación óptima:

```
# .github/workflows/tests.yml
name: Tests

on: [push, pull_request]

jobs:
  tests:
    runs-on: ubuntu-latest
    steps:
```

```
      - uses: actions/checkout@v3

      - name: Setup PHP
        uses: shivammathur/setup-php@v2
        with:
          php-version: 8.3
          coverage: xdebug

      - name: Install Dependencies
        run: composer install

      - name: Run Pint
        run: ./vendor/bin/pint --test

      - name: Run PHPStan
        run: ./vendor/bin/phpstan analyse

      - name: Run Pest
        run: ./vendor/bin/pest --parallel --coverage --min=100
```

Estrategia gradual:

1. Empezar solo con tests
2. Agregar Pint
3. Agregar PHPStan con baseline
4. Subir nivel progresivamente

El error fundamental identificado

Cita clave: "Confundir flexibilidad con libertad"

Laravel es flexible en arquitectura, no en disciplina. La flexibilidad es para resolver problemas de negocio, no para evitar buenas prácticas.

Análisis del error

Manifestaciones comunes:

- Arrays asociativos en lugar de DTOs
- Métodos sin type hints "porque total funciona"
- Tests solo de happy path

- "Lo arreglo después" que nunca llega

Costo real:

- Debugging que consume 40-60% del tiempo de desarrollo
- Bugs en producción que dañan reputación
- Refactorizaciones imposibles sin miedo
- Onboarding lento de nuevos desarrolladores

Roadmap de implementación en proyectos existentes

Fase 1: Evaluación y baseline (Sprint 1-2)

Acciones:

1. Ejecutar PHPStan nivel 5 con baseline
2. Medir cobertura de tipos actual
3. Medir cobertura de tests actual
4. Aplicar Pint una sola vez
5. Documentar estado actual

Entregable: Documento con métricas actuales y brechas.

Fase 2: Quick wins (Sprint 3-4)

Acciones:

1. Configurar Pint en pre-commit hook
2. Implementar CI básico (tests + Pint)
3. Nueva regla: código nuevo debe cumplir 100% tipos y tests
4. Identificar 3 módulos críticos para refactorizar

Entregable: CI funcionando, código nuevo bajo estándares.

Fase 3: Refactoring progresivo (Sprint 5-12)

Acciones:

1. Refactorizar módulos críticos a 100% tipos y tests
2. Subir PHPStan un nivel cada 2 sprints

3. Reducir baseline de PHPStan 10% cada sprint
4. Code reviews enfocados en principios

Entregable: Módulos críticos bajo estándares, PHPStan nivel 7-8.

Fase 4: Consolidación (Sprint 13+)

Acciones:

1. PHPStan nivel MAX sin baseline
2. 100% cobertura en todo el proyecto
3. Mutation testing implementado
4. Documentación de arquitectura

Entregable: Proyecto completamente bajo estándares.

Comparación: Enfoque tradicional vs Good Code

Aspecto	Tradicional	Good Code
Types	Opcional	Obligatorio 100%
Tests	"Lo importante"	100% exacto
Análisis estático	Si da tiempo	PHPStan MAX
Formato	Discutido en CR	Automatizado
CI	Tests básicos	Guardián completo
Velocidad inicial	Rápida	Moderada
Velocidad sostenida	Decreciente	Creciente
Confianza deployment	Rezar	Saber

Relación con SOLID

Single Responsibility Principle

Clases con una responsabilidad son más fáciles de tipar y testear al 100%.

Open/Closed Principle

Interfaces tipadas garantizan extensión sin modificación.

Liskov Substitution Principle

PHPStan MAX detecta violaciones automáticamente.

Interface Segregation Principle

Cobertura de tipos fuerza interfaces específicas.

Dependency Inversion Principle

Type hints en constructores formalizan inversión de dependencias.

Consideraciones finales

Pregunta clave

¿Es esto overkill para proyectos pequeños?

Respuesta: Depende del horizonte temporal. Si el proyecto vivirá más de 6 meses, no es overkill. Es inversión.

Obstáculos reales

1. **Resistencia del equipo**: Cambio cultural requiere tiempo
2. **Tiempo inicial**: Primera implementación consume sprints
3. **Falsa sensación de lentitud**: Strictness se siente lenta hasta que debuggear se vuelve raro

Beneficios medibles

- Reducción de bugs en producción: 60-80%
- Tiempo de onboarding: -50%
- Confianza en refactoring: +200%
- Velocidad de features (después de 3 meses): +30%

Conclusión

La filosofía de Good Code no es una lista de herramientas. Es un cambio de mentalidad: de "funciona ahora" a "funciona siempre".

El starter kit de Nuno Maduro no es el objetivo. Es el ejemplo de que es posible. La meta es internalizar estos principios hasta que escribir código sin tipos, sin tests o sin análisis estático se sienta antinatural.

Tesis final: Strictness no es burocracia. Es la única forma conocida de escalar complejidad sin colapsar en caos.

Recursos

- Laravel Starter Kit de Nuno Maduro
- PHPStan Documentation
- Pest PHP Documentation
- Laravel Pint Documentation
- Principios SOLID aplicados a Laravel - Opinated
- Agradecimientos a Vishal Rajpurohit por su hilo conductor

Aviso