Informe final — Diseño y prácticas para módulos IaC

Este informe sintetiza las decisiones de diseño, el análisis de patrones (Facade, Adapter, Mediator), la comparativa entre monorepositorio y multirepositorio, y las prácticas de versionado y publicación de módulos Terraform desarrolladas durante las fases del laboratorio.

1. **Análisis de patrones (Facade, Adapter, Mediator)**
   - Facade
     - Propósito: ofrecer una interfaz simplificada y estable a un subsistema complejo (aquí: red + servidor + firewall).
     - Acoplamiento / reutilización: reduce el acoplamiento del consumidor con la implementación interna; favorece reutilización porque el consumidor depende de un contrato (outputs estables) y no de detalles internos.
     - Ventaja: encapsula la orquestación; permite cambios internos sin romper consumidores si los outputs se mantienen.
     - Inconveniente: el facade puede volverse un punto único de versión y obligar a coordinar cambios internos.
   - Adapter
     - Propósito: traducir entre interfaces incompatibles (formatos/contratos distintos).
     - Acoplamiento / reutilización: desacopla consumidores de proveedores/legacy; facilita reutilizar módulos existentes adaptando su salida al formato esperado.
     - Ventaja: permite integrar rápidamente sistemas heterogéneos; es ideal para migraciones y compatibilidad backwards.
     - Inconveniente: si el adapter realiza lógica compleja, puede ocultar errores o causar duplicación de lógica.
   - Mediator
     - Propósito: centralizar la coordinación y las reglas de interacción entre múltiples componentes.
     - Acoplamiento / reutilización: rompe dependencias punto-a-punto; los componentes solo conocen al mediador.
     - Ventaja: excelente para orquestación compleja donde las relaciones entre módulos cambian frecuentemente.
     - Inconveniente: puede convertirse en un "god object" con mucha lógica; requiere tests y límites claros.

- Comparativa rápida
    - Si buscas simplicidad de uso para consumidores: **Facade**.
    - Si necesitas compatibilidad con APIs diferentes: **Adapter**.
    - Si necesitas coordinar interacciones complejas entre varios módulos: **Mediator**.

2. **Elecciones de diseño en el laboratorio:**
    - Separación unidireccional (Fase 1): la red publica outputs (archivo JSON + Terraform outputs) y los consumidores(servidor) los leen, esta elección favorece claridad en responsabilidades y pruebas unitarias de cada modulo.
    - Inyección de dependencias (Fase 2): **main.py** paso a inyectar configuración adicional del servidor (server_config) en **triggers**, lo que demostró cómo parametrizar recursos sin acoplar código de negocio a la implementación de la red.
    - Facade (Fase 3): se creó **facade.tf.json** para exponer **bucket_name**, **endpoint**, **network_id**. Refactorizar los scripts para leer el **Facade** reduce el acoplamiento y centraliza el contrato público.

3. **Versionado y publicación (Fase 8 y 9):**

- Uso recomendado: SemVer (MAJOR.MINOR.PATCH) para módulos Terraform.
    - MAJOR cuando rompes outputs o la interfaz del módulo.
    - MINOR para nuevas funcionalidades compatibles (ej. nuevo output no obligatorio).
    - PATCH para correcciones y ajustes internos.
- Consecuencias de omitir SemVer:
    - Fricción en consumidores: actualizaciones inesperadas pueden romper pipelines.
    - Mayor esfuerzo en diagnóstico y rollback.
- Política de releases sugerida:
    - **Cadencia**: releases a demanda; parches frecuentes, minors por conjunto de features estable.
    - **Criterios**: pruebas automáticas verdes, revisión de compatibilidad.
    - **Proceso**: PR → CI (tests) → aprobar → tag (make release VERSION=x.y.z) → push tags → publicar artifact en registry.

4. **Publicación: registry local vs Terraform Cloud Registry**

- **Registry local (ventajas):** control corporativo, integración con Artifactory/Harbor, políticas de acceso.(desventaja) requiere mantenimiento y configuración de mirrors/CLI.
- **Terraform Cloud Registry (ventajas):** integrado con Terraform Cloud, UI, búsqueda, gestión de versiones. (desventaja) puede implicar costos y dependencias externas.

5. **Flujo Git y gestión de versiones**

- Propuesta para monorepo:
    - Rama principal **main**; ramas de feature **feature/x**; PRs con scope claro.
    - **Versionado:** tags por modulo usando convención **module-name/vX.Y.Z** o registro externo con artefactos.
    - **CI:** pipelines por affected files (tests sólo para módulos modificados), checkout monorepo.
- Propuesta para multirepo:
    - Cada modulo tiene su repo, su pipeline y su release independent.
    - Cambios cross-cutting: coordinación mediante PRs encadenados y un plan de integración.

6. **Ejercicios adicionales (respuestas y propuestas)**
    - Diferencias clave entre Facade, Adapter y Mediator (acoplamiento y reutilización):
        - Facade reduce acoplamiento del consumidor al ocultar subsistema; reutilización alta si el facade mantiene contratos.
        - Adapter reduce acoplamiento entre implementaciones incompatibles; facilita reutilización sin cambiar consumidores.
        - Mediator reduce acoplamiento entre componentes internos a costa de centralizar la lógica de interacción.
    - Escenario real: despliegue multi-cloud
        - **Problema**: distintas VPC/nombres de recursos y requisitos por proveedor.
        - **Opción 1:** Facade que expone outputs neutrales (endpoint, network_id, region) y oculta diferencias internas por proveedor. Ventaja: consumidores no cambian. Inconveniente: el facade debe mapear muchas diferencias.
        - **Opción 2:** Adapter por proveedor que normalice outputs a la interfaz común; el Mediator podría coordinar cross-cloud workflows.

- IoC y Dependency Inversion en IaC
    - **Inversión de Control:** los módulos no crean sus dependencias, las reciben (outputs o inyección). Mejora testabilidad y desacoplamiento.
    - **Inversión de Dependencias:** los módulos altos dependen de abstracciones (outputs estables) en lugar de implementaciones concretas. Mejora mantenibilidad porque los cambios se localizan.

- Riesgos/anti-patrones de abusar de inyección de dependencias en Terraform
    - Exponer demasiada configuración en triggers puede crear superficies inestables (recreaciones frecuentes).
    - Over-inyección: pasar estructuras complejas (JSON extenso) en triggers puede desordenar el estado y complicar diffs.
    - Antídotos: documentar contratos, validar formatos, usar JSON con keys estables y versionadas.
- Migración a multi-repositorio
    - Inventario de módulos, crear repos por módulo, extraer carpeta con historial mínimo (git subtree or filter-repo), configurar CI por repo, publicar artefactos a registry.

7. Conclusión

Las decisiones tomadas en el laboratorio (separación unidireccional, uso de Facade/Adapter/Mediator según caso, y versionado SemVer) permiten construir una arquitectura IaC más mantenible y escalable. Las prácticas recomendadas incluyen documentar contratos de outputs, automatizar releases y testear adaptadores y mediadores. Para obtener resiliencia y gobernanza en entornos corporativos, combinar registro privado, políticas de acceso y pipelines de CI/CD por módulo ofrece el mejor equilibrio entre autonomía y control.