Guía Práctica: Static Code Analysis y Code Review con SonarQube

SonarQube es una herramienta de revisión automática de código que aplica análisis estático para detectar problemas de calidad en etapas tempranas del ciclo de desarrollo. Además de seguridad, permite trabajar de forma sistemática sobre **mantenibilidad**, **fiabilidad**, **deuda técnica**, **duplicación**, **code smells** y **bugs**.

Objetivo

Esta práctica te permitirá montar un flujo de *code review* asistido por análisis estático en un repositorio GitHub, desplegando SonarQube con Docker e integrándolo con GitHub Actions para validar cambios en cada *push* y *pull request*.

Requisitos Previos

- Conocimiento básico de Git, GitHub y flujos con *pull requests*.
- Docker y Docker Compose instalados.
- Java 17+ y Maven (para los ejemplos prácticos en Java).
- Una cuenta de GitHub y permisos para crear secretos del repositorio.

---

1. Conceptos clave de static code analysis para code review

1.1 Revisión manual + análisis automatizado

El *code review* manual sigue siendo necesario, pero debe complementarse con análisis estático automatizado para detectar problemas repetitivos y objetivos:

- Duplicación de código.
- Complejidad cognitiva alta.
- Violaciones de convenciones de diseño.
- Errores potenciales (*bugs*).

- Problemas de mantenibilidad (*code smells*).

1.2 El análisis estático no sustituye al diseño

El análisis del código fuente ayuda mucho, pero no cubre por sí solo todos los problemas del sistema. También hay que revisar diseño y arquitectura. En esta práctica nos centramos en la parte de construcción y revisión de código.

1.3 Mantenibilidad y calidad técnica

En construcción de software, mejorar mantenibilidad reduce errores y acelera el cambio. Esto se trabaja con:

- Refactoring incremental.
- Reducción de duplicación (principio DRY).
- Simplificación de lógica compleja.
- Estandarización de estilo y reglas de calidad.

1.4 SonarQube como herramienta de code review continuo

SonarQube permite evaluar el código en términos de:

- **Maintainability**.
- **Reliability**.
- **Security**.

Y detectar *issues* como duplicación, complejidad excesiva, *code smells*, *bugs*, vulnerabilidades y *security hotspots*.

---

2. Preparación del repositorio GitHub (práctica DevOps)

Crea un repositorio para la práctica, por ejemplo: `sonarqube-lab`.

```
git clone <URL_DEL_REPO>
cd sonarqube-lab
```

Si necesitas un proyecto Java base para la práctica:

```
mvn archetype:generate -DgroupId=org.uca.gii \
  -DartifactId=sonarqube-lab \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DinteractiveMode=false
```

Buenas prácticas DevOps desde el inicio:

- Trabajar por ramas cortas (`feature/...`).
- Abrir *pull requests* pequeñas.
- Ejecutar análisis automático en cada PR.
- No hacer merge si falla el *quality gate*.

---

3. Despliegue de SonarQube con Docker

Crea `infra/sonarqube/docker-compose.yml` :

```
services:
  sonarqube:
    image: sonarqube:lts-community
    container_name: sonarqube
    depends_on:
      - db
    ports:
      - "9000:9000"
    environment:
      SONAR_JDBC_URL: jdbc:postgresql://db:5432/sonarqube
      SONAR_JDBC_USERNAME: sonarqube
      SONAR_JDBC_PASSWORD: sonarqube
    volumes:
      - sonarqube_data:/opt/sonarqube/data
      - sonarqube_logs:/opt/sonarqube/logs
      - sonarqube_extensions:/opt/sonarqube/extensions

  db:
    image: postgres:15
    container_name: sonarqube_db
    environment:
      POSTGRES_USER: sonarqube
      POSTGRES_PASSWORD: sonarqube
      POSTGRES_DB: sonarqube
    volumes:
      - postgresql:/var/lib/postgresql/data

volumes:
  sonarqube_data:
  sonarqube_logs:
  sonarqube_extensions:
  postgresql:
```

Arranque y parada:

```
docker compose -f infra/sonarqube/docker-compose.yml up -d
docker compose -f infra/sonarqube/docker-compose.yml ps
docker compose -f infra/sonarqube/docker-compose.yml down
```

Accede a http://localhost:9000 y cambia credenciales por defecto (`admin/admin`).

---

4. Crear proyecto y token en SonarQube

1. Crea un proyecto local en SonarQube (ejemplo: `uca-gii-sonarqube-lab`).
2. Genera un token de análisis.
3. En GitHub, configura secretos del repositorio:

- `SONAR_HOST_URL` = URL accesible del servidor SonarQube.
- `SONAR_TOKEN` = token generado en SonarQube.

> Si usas SonarQube dockerizado en tu máquina, GitHub Actions debe ejecutarse en un **runner self-hosted** con conectividad a ese servidor.

---

5. Integración con GitHub Actions

Crea `.github/workflows/sonarqube.yml` :

```yaml
name: SonarQube Quality

on:
  push:
    branches: ["main"]
  pull_request:

jobs:
  analyze:
    runs-on: [self-hosted, linux]

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Java
        uses: actions/setup-java@v4
        with:
          distribution: temurin
          java-version: '17'
          cache: maven

      - name: Build and tests
        run: mvn -B -ntp verify

      - name: SonarQube analysis + Quality Gate
        run: >
          mvn -B -ntp sonar:sonar
          -Dsonar.projectKey=uca-gii-sonarqube-lab
          -Dsonar.host.url=${{ secrets.SONAR_HOST_URL }}
          -Dsonar.token=${{ secrets.SONAR_TOKEN }}
          -Dsonar.qualitygate.wait=true
```

Recomendación:

- Protege `main` y exige el *check* de este workflow antes de hacer merge.

---

6. Ejercicios prácticos (propuestos y resueltos)

Los ejercicios se basan en ejemplos de `slides/implementacion/implementacion.md` (refactoring, duplicación, cohesión y complejidad).

Dinámica DevOps para cada ejercicio

1. Crear rama: `feature/eX-...`.
2. Subir versión inicial (con problemas) y abrir PR.
3. Revisar hallazgos en SonarQube.
4. Aplicar refactoring en commits pequeños.
5. Verificar *quality gate* en verde.
6. Hacer merge.

Ejercicio 1: Duplicación y diseño descohesionado (nóminas)

Propuesta

Partir de una versión con responsabilidades mezcladas y señales de diseño poco cohesionadas (caso `Empleado` / `Autonomo`).

```
public class Empleado {
  Comparable id;
  String name;
  float yearlyGrossSalary;

  public float computeMonthlySalary() {
    return yearlyGrossSalary / 12;
  }
}

public class Autonomo extends Empleado {
  String vatCode;
  float workingHours;

  public float computeMonthlySalary() {
    return workingHours * Company.getHourlyRate() * (1.0f + Company.getVatRa
  }
}
```

Hallazgos esperados en SonarQube

- Problemas de mantenibilidad por diseño poco cohesionado.
- Duplicación en estructura y construcción de entidades.
- *Code smells* por encapsulación insuficiente y responsabilidades mezcladas.

Resolución

Refactorizar a jerarquía cohesionada con contrato explícito (`Empleado` abstracto + especializaciones):

```
public abstract class Empleado {
  protected Comparable id;
  protected String name;

  public Empleado(String id, String name) {
    this.id = id;
    this.name = name;
  }

  public abstract float computeMonthlySalary();
}

public class Plantilla extends Empleado {
  private float yearlyGrossSalary;

  public Plantilla(String id, String name, float yearlyGrossSalary) {
    super(id, name);
    this.yearlyGrossSalary = yearlyGrossSalary;
  }

  @Override
  public float computeMonthlySalary() {
    return yearlyGrossSalary / 12;
  }
}

public class Autonomo extends Empleado {
  private String vatCode;
  private float workingHours;

  public Autonomo(String id, String name, String vatCode) {
    super(id, name);
    this.vatCode = vatCode;
  }

  public void addWorkingHours(float hours) {
    this.workingHours += hours;
```

```
  }

  @Override
  public float computeMonthlySalary() {
    return workingHours * Company.getHourlyRate() * (1.0f + Company.getVatRa
  }
}
```

Resultado esperado: mejora de mantenibilidad y reducción de *smells*.

Ejercicio 2: Complejidad cognitiva y duplicación en manejo de errores

Propuesta

Analizar una versión con anidamiento excesivo y lógica de errores duplicada:

```
if (deletePage(page) == E_OK) {
  if (registry.deleteReference(page.name) == E_OK) {
    if (configKeys.deleteKey(page.name.makeKey()) == E_OK) {
      logger.log("page deleted");
    } else {
      logger.log("configKey not deleted");
    }
  } else {
    logger.log("deleteReference from registry failed");
  }
} else {
  logger.log("delete failed");
  return E_ERROR;
}
```

Hallazgos esperados en SonarQube

- Complejidad cognitiva alta.
- Anidamiento profundo.
- Duplicación de patrones de logging/gestión de error.

Resolución

Separar lógica principal y tratamiento de errores usando excepciones y extracción de método:

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  } catch (Exception e) {
    logError(e);
  }
}

private void deletePageAndAllReferences(Page page) throws Exception {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
  logger.log(e.getMessage());
}
```

Resultado esperado: reducción de complejidad, mejor legibilidad y menor deuda técnica.

Ejercicio 3: Proceso DevOps completo en GitHub

Propuesta

Aplicar el flujo anterior en un repositorio público/privado:

1. PR con código inicial (esperable que falle *quality gate*).
2. PR actualizada con refactor.
3. Validar en SonarQube que bajan duplicación y *smells*.
4. Merge solo tras estado verde del workflow.

Resolución esperada

- Historial de commits pequeño y trazable.

- Evidencia en PR de antes/después.
- Rama `main` protegida por análisis automático.

---

7. Checklist de buenas prácticas

- SonarQube como *quality gate* en PR, no solo análisis manual ocasional.
- Cambios pequeños y frecuentes.
- Refactoring separado de cambios funcionales.
- Métricas de calidad tratadas como criterio de entrega.
- Secretos fuera del código (`SONAR_TOKEN`, `SONAR_HOST_URL`).
- Infraestructura versionada (Docker Compose en el repo).

Ejercicio Entregable

1. Selecciona un repositorio GitHub con código real (Java o multi-lenguaje).
2. Despliega SonarQube con Docker.
3. Integra el análisis en GitHub Actions con *runner self-hosted*.
4. Ejecuta al menos dos iteraciones de mejora (*baseline* + refactor).
5. Entrega un informe en `README.md` con:

- URL del repositorio.
- Flujo de despliegue y CI configurado.
- Hallazgos principales (bugs, *code smells*, duplicación, vulnerabilidades si aparecen).
- Refactors aplicados y su impacto en métricas de calidad.