java\_analyzer.py all files in test\_samples folder to find relationship -- inhettence, association, composition, aggregation , implement

Collecting workspace informationThe java\_analyzer.py script is designed to analyze all `.java` files in the test\_samples folder and extract class relationships, including:

- \*\*Inheritance\*\* (class extends another class)

- \*\*Association\*\* (uses another class as a parameter, field, or local variable)

- \*\*Composition\*\* (has a field of another class type)

- \*\*Aggregation\*\* (has a collection or plural field of another class type)

- \*\*Implementation\*\* (class implements an interface)

It does this by:

1. Scanning all `.java` files in test\_samples using `find\_java\_files`.

2. Extracting class names, inheritance, and interface implementations with `analyze\_java\_file`.

3. Extracting fields, associations, and compositions with `analyze\_java\_fields\_and\_associations`.

4. Optionally, extracting Spring annotations with `analyze\_spring\_annotations`.

5. Visualizing the relationships with `visualize\_relationships`.

To run the analysis and generate a diagram, simply execute:

```sh

python test\_samples/java\_analyzer.py

```

This will print the relationships and generate a `class\_diagram.png` file showing the relationships between classes in test\_samples.

output

(v11) PS E:\PYTHON\_PROJECTS\Python-Utilities\class\_viz\_dashboard> python test\_samples/java\_analyzer.py

{'aggregation': [],

'association': [('UserRepository', 'User', 'user'),

('UserService', 'EmailService', 'emailService'),

('UserService', 'Order', 'order'),

('UserService', 'User', 'user'),

('UserService', 'UserRepository', 'userRepository')],

'composition': [('UserService', 'Order', 'order'),

('UserService', 'UserRepository', 'userRepository')],

'inheritance': [('UserRepository', 'BaseRepository'),

('UserService', 'BaseService')],

'interface\_implementation': [('UserService', 'UserApi')],

'strong\_composition': []}

--- Relationships with variable names ---

UserService composes Order via variable 'order'

UserService composes UserRepository via variable 'userRepository'

UserRepository associates User via variable 'user'

UserService associates EmailService via variable 'emailService'

UserService associates Order via variable 'order'

UserService associates User via variable 'user'

UserService associates UserRepository via variable 'userRepository'

UserRepository inherits BaseRepository

UserService inherits BaseService

UserService implements UserApi

--- Spring Annotations ---

Class 'EmailService' is annotated with @Service

Class 'UserRepository' is annotated with @Repository

Class 'UserService' is annotated with @Service

Field 'userRepository' of type 'UserRepository' is @Autowired in test\_samples\UserService.java

Constructor of 'UserService' is @Autowired with params (UserRepository userRepository, EmailService emailService) in test\_samples\UserService.java

Diagram saved to class\_diagram.png

can you verify everthing correct ..are we getting all realtionships also variable which creates relationship

In the context of software engineering and programming, \*\*"model"\*\* can have several meanings depending on the context. Here are the most common uses:

---

### 1. \*\*Data Model / Domain Model\*\*

- A \*\*model\*\* often refers to a class or set of classes that represent the data and business logic of your application.

- In Java or Python, a model class typically represents an entity (like `User`, `Order`, `Product`) and contains fields (attributes) and sometimes methods related to that entity.

\*\*Example (Java):\*\*

```java

public class User {

private String name;

private String email;

// getters, setters, etc.

}

```

\*\*Example (Python):\*\*

```python

class User:

def \_\_init\_\_(self, name, email):

self.name = name

self.email = email

```

---

### 2. \*\*MVC Pattern (Model-View-Controller)\*\*

- In the \*\*MVC\*\* architectural pattern, the \*\*Model\*\* is responsible for managing the data, logic, and rules of the application.

- The \*\*View\*\* displays the data, and the \*\*Controller\*\* handles user input.

---

### 3. \*\*Database Model\*\*

- In frameworks like Django (Python) or Spring (Java), a \*\*model\*\* often maps to a database table.

- Each instance of the model represents a row in the table.

---

### 4. \*\*Machine Learning Model\*\*

- In machine learning, a \*\*model\*\* is a mathematical representation trained to make predictions or decisions based on data.

---

## \*\*Summary\*\*

- In your context (Java class diagrams and relationships), a \*\*model\*\* is most likely a class that represents a real-world entity or concept in your application, often used to structure and manage data.

If you have a specific context (e.g., Spring, Django, ML), let me know for a more tailored explanation!

Certainly! Here’s a `groovy\_analyzer.py` script similar in structure to your `java\_analyzer.py`, but adapted for Groovy syntax. It will scan all `.groovy` files in a given directory, extract class relationships (inheritance, association, composition, aggregation, implementation), and visualize them.

````python

import os

import re

from graphviz import Digraph

RELATIONSHIP\_KEYS = [

'aggregation',

'composition',

'strong\_composition',

'association',

'inheritance',

'interface\_implementation'

]

def find\_groovy\_files(root\_dir):

groovy\_files = []

for dirpath, \_, filenames in os.walk(root\_dir):

for fname in filenames:

if fname.endswith('.groovy'):

groovy\_files.append(os.path.join(dirpath, fname))

return groovy\_files

def analyze\_groovy\_file(filepath, relationships, class\_names):

with open(filepath, "r", encoding="utf-8") as f:

source = f.read()

# Remove comments

source = re.sub(r'//.\*?$|/\\*.\*?\\*/', '', source, flags=re.DOTALL | re.MULTILINE)

# Find all class/interface definitions

class\_pattern = re.compile(

r'(class|interface)\s+(\w+)(?:\s+extends\s+(\w+))?(?:\s+implements\s+([\w,\s]+))?', re.MULTILINE)

classes = class\_pattern.findall(source)

for kind, class\_name, parent, interfaces in classes:

class\_names.add(class\_name)

if parent:

relationships['inheritance'].append((class\_name, parent))

if interfaces:

for iface in [i.strip() for i in interfaces.split(',') if i.strip()]:

relationships['interface\_implementation'].append((class\_name, iface))

def analyze\_groovy\_fields\_and\_associations(filepath, relationships, class\_names):

with open(filepath, "r", encoding="utf-8") as f:

source = f.read()

# Remove comments

source = re.sub(r'//.\*?$|/\\*.\*?\\*/', '', source, flags=re.DOTALL | re.MULTILINE)

# Groovy allows def and type inference, but we'll look for explicit types and def

class\_body\_pattern = re.compile(

r'(class|interface)\s+(\w+)[^{]\*\{((?:[^{}]|\{[^{}]\*\})\*)}', re.DOTALL)

for match in class\_body\_pattern.finditer(source):

kind, class\_name, body = match.groups()

# Fields (composition/aggregation)

field\_pattern = re.compile(r'(def|\w+)\s+(\w+)\s\*(=|;)', re.MULTILINE)

for field\_type, var\_name, \_ in field\_pattern.findall(body):

if field\_type in class\_names:

if re.search(r'List<\s\*' + field\_type + r'\s\*>', body) or var\_name.endswith('s'):

relationships['aggregation'].append((class\_name, field\_type, var\_name))

else:

relationships['composition'].append((class\_name, field\_type, var\_name))

# Method parameters (association)

method\_pattern = re.compile(r'(?:def|\w+)\s+\w+\s\*\(([^)]\*)\)', re.MULTILINE)

for params in method\_pattern.findall(body):

for param in params.split(','):

param = param.strip()

if not param:

continue

parts = param.split()

if len(parts) == 2:

param\_type, param\_name = parts

if param\_type in class\_names:

relationships['association'].append((class\_name, param\_type, param\_name))

# Local variables in methods (association)

local\_var\_pattern = re.compile(r'(def|\w+)\s+(\w+)\s\*=\s\*new\s+(\w+)\s\*\(', re.MULTILINE)

for var\_type, var\_name, new\_type in local\_var\_pattern.findall(body):

if new\_type in class\_names:

relationships['association'].append((class\_name, new\_type, var\_name))

def visualize\_relationships(relationships, output\_file='groovy\_class\_diagram'):

dot = Digraph(comment='Groovy Class Diagram', format='png')

dot.attr(rankdir='LR')

# Collect all class names

class\_names = set()

for rels in relationships.values():

for rel in rels:

class\_names.add(rel[0])

class\_names.add(rel[1])

# Add nodes for each class

for cls in class\_names:

dot.node(cls, cls, shape='box')

# Add edges for each relationship type, showing variable name if present

for a, b, var in relationships.get('composition', []):

dot.edge(a, b, label=f"{var} (composes)", arrowhead='diamond', color='green')

for a, b, var in relationships.get('strong\_composition', []):

dot.edge(a, b, label=f"{var} (strong comp.)", arrowhead='diamond', color='red', penwidth='2')

for a, b, var in relationships.get('aggregation', []):

dot.edge(a, b, label=f"{var} (aggregates)", arrowhead='odiamond', color='orange')

for a, b, var in relationships.get('association', []):

dot.edge(a, b, label=f"{var} (associates)", style='dotted', color='purple')

for a, b in relationships.get('inheritance', []):

dot.edge(b, a, label='inherits', arrowhead='onormal', color='black')

for a, b in relationships.get('interface\_implementation', []):

dot.edge(b, a, label='implements', style='dashed', color='blue')

dot.render(output\_file, view=True)

print(f"Diagram saved to {output\_file}.png")

def main():

relationships = {k: [] for k in RELATIONSHIP\_KEYS}

class\_names = set()

groovy\_files = find\_groovy\_files('test\_samples')

# First pass: collect all class names and inheritance/interfaces

for filename in groovy\_files:

try:

analyze\_groovy\_file(filename, relationships, class\_names)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

# Second pass: collect fields and associations

for filename in groovy\_files:

try:

analyze\_groovy\_fields\_and\_associations(filename, relationships, class\_names)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

# Remove duplicates and sort for consistent output

for k in relationships:

relationships[k] = sorted(list(set(relationships[k])))

import pprint

pprint.pprint(relationships)

print("\n--- Relationships with variable names ---")

for rel in relationships['composition']:

print(f"{rel[0]} composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['strong\_composition']:

print(f"{rel[0]} strong composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['aggregation']:

print(f"{rel[0]} aggregates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['association']:

print(f"{rel[0]} associates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['inheritance']:

print(f"{rel[0]} inherits {rel[1]}")

for rel in relationships['interface\_implementation']:

print(f"{rel[0]} implements {rel[1]}")

visualize\_relationships(relationships)

if \_\_name\_\_ == "\_\_main\_\_":

main()

````

\*\*How to use:\*\*

1. Place this script in your project (e.g., `test\_samples/groovy\_analyzer.py`).

2. Make sure you have some `.groovy` files in the test\_samples folder.

3. Run the script:

```sh

python test\_samples/groovy\_analyzer.py

```

4. It will print relationships and generate a `groovy\_class\_diagram.png` file.

\*\*Note:\*\*

Groovy is more dynamic than Java, so static analysis may miss some relationships, especially with `def` and dynamic typing. For best results, use explicit types in your Groovy classes.

this is output is this ok

(v11) PS E:\PYTHON\_PROJECTS\Python-Utilities\class\_viz\_dashboard> python test\_samples/java\_analyzer.py

{'aggregation': [],

'association': [('UserRepository', 'User', 'user'),

('UserService', 'EmailService', 'emailService'),

('UserService', 'Order', 'order'),

('UserService', 'User', 'user'),

('UserService', 'UserRepository', 'userRepository')],

'composition': [('UserService', 'Order', 'order'),

('UserService', 'UserRepository', 'userRepository')],

'inheritance': [('UserRepository', 'BaseRepository'),

('UserService', 'BaseService')],

'interface\_implementation': [('UserService', 'UserApi')],

'strong\_composition': []}

--- Relationships with variable names ---

UserService composes Order via variable 'order'

UserService composes UserRepository via variable 'userRepository'

UserRepository associates User via variable 'user'

UserService associates EmailService via variable 'emailService'

UserService associates Order via variable 'order'

UserService associates User via variable 'user'

UserService associates UserRepository via variable 'userRepository'

UserRepository inherits BaseRepository

UserService inherits BaseService

UserService implements UserApi

--- Spring Annotations ---

Class 'EmailService' is annotated with @Service

Class 'UserRepository' is annotated with @Repository

Class 'UserService' is annotated with @Service

Field 'userRepository' of type 'UserRepository' is @Autowired in test\_samples\UserService.java

Constructor of 'UserService' is @Autowired with params (UserRepository userRepository, EmailService emailService) in test\_samples\UserService.java

Yes, your output looks correct and matches the expected relationships for a typical Java Spring project. Here’s a quick verification:

- \*\*Inheritance:\*\*

- `UserRepository` inherits `BaseRepository`

- `UserService` inherits `BaseService`

- \*\*Interface Implementation:\*\*

- `UserService` implements `UserApi`

- \*\*Composition:\*\*

- `UserService` composes `Order` via `order`

- `UserService` composes `UserRepository` via `userRepository`

- \*\*Association:\*\*

- `UserRepository` associates `User` via `user`

- `UserService` associates `EmailService`, `Order`, `User`, and `UserRepository` via respective variables

- \*\*Spring Annotations:\*\*

- Detected `@Service`, `@Repository`, and `@Autowired` annotations correctly

- \*\*No aggregation or strong composition\*\* detected, which is fine if your codebase doesn’t use those patterns.

\*\*Conclusion:\*\*

Your analyzer is correctly extracting and displaying class relationships and variable names that create those relationships. The output is as expected for the provided code. If you add more classes or relationships, they should appear in the output as well.

want your suggestion -- from the java\_analyzer.py if we store in database .... some developer is changing code ...... based on saved realtionship ..... can we suggest him ....due to your changes in this file ...it will effect these all reatlions ....... a kind of warning .... this is sample code only .. imagine we have production code with huge files and logic

This is an excellent idea and a common requirement in large codebases—\*\*impact analysis\*\*. Here’s a high-level approach and suggestions for implementing this:

---

## \*\*How It Could Work\*\*

1. \*\*Store Relationships in a Database\*\*

- After each analysis run, save all extracted relationships (inheritance, association, etc.) in a database table, including the source file and line numbers if possible.

2. \*\*Detect Code Changes\*\*

- Use a version control system (like Git) to detect which files have changed.

- Alternatively, monitor file modification times or use a pre-commit hook.

3. \*\*Compare Old and New Relationships\*\*

- Re-run the analyzer on changed files.

- Compare the new relationships with those stored in the database.

- Identify what relationships have been added, removed, or changed.

4. \*\*Impact Analysis\*\*

- For each changed relationship, find all related classes/files from the database.

- Generate a report or warning:

\_"Changing `UserService.groovy` affects these relationships: [list]. These classes may be impacted: [list]."\_

5. \*\*Notify Developer\*\*

- Show this warning in the IDE, CI/CD pipeline, or as a report.

---

## \*\*Sample Workflow\*\*

1. \*\*Initial Analysis\*\*

- Run analyzer, store relationships in DB:

```

| ClassA | inherits | ClassB | file: ClassA.groovy | line: 5 |

| ... | ... | ... | ... | ... |

```

2. \*\*Developer Changes `ClassA.groovy`\*\*

- Analyzer detects `ClassA` now no longer inherits `ClassB`.

3. \*\*Impact Report\*\*

- System finds all classes that depend on `ClassA`'s inheritance.

- Warns:

\_"Removing inheritance from `ClassA` may break: [ClassC, ClassD] which depend on this relationship."\_

---

## \*\*Benefits\*\*

- \*\*Proactive Warnings:\*\* Developers see the impact before merging code.

- \*\*Better Code Quality:\*\* Reduces accidental breaking changes.

- \*\*Documentation:\*\* Keeps a living map of code relationships.

---

## \*\*Implementation Tips\*\*

- Use a relational database (PostgreSQL, MySQL, SQLite) for storing relationships.

- Store: class name, relationship type, related class, variable name, file, line number, timestamp.

- Use Git hooks or CI/CD to trigger analysis on code changes.

- For large codebases, optimize by analyzing only changed files and their dependencies.

---

## \*\*Summary\*\*

Yes, you can absolutely build a system that warns developers about the impact of their changes by storing and comparing class relationships. This is a valuable tool for large, complex projects and can be integrated into your development workflow for better maintainability and safety.2. \*\*Developer Changes `ClassA.groovy`\*\*

- Analyzer detects `ClassA` now no longer inherits `ClassB`.

3. \*\*Impact Report\*\*

- System finds all classes that depend on `ClassA`'s inheritance.

- Warns:

\_"Removing inheritance from `ClassA` may break: [ClassC, ClassD] which depend on this relationship."\_

---

## \*\*Benefits\*\*

- \*\*Proactive Warnings:\*\* Developers see the impact before merging code.

- \*\*Better Code Quality:\*\* Reduces accidental breaking changes.

- \*\*Documentation:\*\* Keeps a living map of code relationships.

---

## \*\*Implementation Tips\*\*

- Use a relational database (PostgreSQL, MySQL, SQLite) for storing relationships.

- Store: class name, relationship type, related class, variable name, file, line number, timestamp.

- Use Git hooks or CI/CD to trigger analysis on code changes.

- For large codebases, optimize by analyzing only changed files and their dependencies.

---

## \*\*Summary\*\*

Yes, you can absolutely build a system that warns developers about the impact of their changes by storing and comparing class relationships. This is a valuable tool for large, complex projects and can be integrated into your development workflow for better maintainability and safety.

can we create such model uding java\_analyzer.py .....impact analysis .. and test localy

Yes, you \*\*can\*\* create a model for impact analysis using your java\_analyzer.py! Here’s how you can approach it and test locally:

---

## \*\*1. Store Relationships as a Model\*\*

After running your analyzer, save the relationships (e.g., as a JSON file or in a SQLite database).

Example for JSON:

```python

import json

# After relationships are built

with open('relationships\_snapshot.json', 'w') as f:

json.dump(relationships, f, indent=2)

```

---

## \*\*2. Detect Code Changes\*\*

- Use file modification times, or

- Use `git diff` to see which files have changed.

---

## \*\*3. Re-analyze and Compare\*\*

After a code change:

- Run java\_analyzer.py again to get the new relationships.

- Load the previous snapshot and compare.

Example comparison logic:

```python

with open('relationships\_snapshot.json') as f:

old\_relationships = json.load(f)

# Assume `relationships` is the new one

for rel\_type in RELATIONSHIP\_KEYS:

removed = set(old\_relationships[rel\_type]) - set(relationships[rel\_type])

added = set(relationships[rel\_type]) - set(old\_relationships[rel\_type])

if removed:

print(f"Removed {rel\_type}: {removed}")

if added:

print(f"Added {rel\_type}: {added}")

```

---

## \*\*4. Impact Analysis Output\*\*

For each removed or added relationship, print or log a warning:

- “Changing `UserService.java` removed inheritance from `BaseService`. This may affect: [list classes that depended on this].”

---

## \*\*5. Test Locally\*\*

1. Run java\_analyzer.py and save the snapshot.

2. Make a code change.

3. Run java\_analyzer.py again and compare.

4. See the printed warnings about impacted relationships.

---

## \*\*Summary\*\*

- \*\*Yes, you can do this locally.\*\*

- Use your analyzer to snapshot relationships.

- After changes, compare snapshots and print impact warnings.

- You can use JSON or SQLite for storage.

If you want, I can provide a ready-to-use code snippet for the snapshot/compare logic!

pls. provide a ready-to-use code smippet for the snapshot/compare

Certainly! Here’s a \*\*ready-to-use code snippet\*\* you can add to your java\_analyzer.py for snapshotting and comparing relationships for impact analysis.

---

#### 1. \*\*Add these imports at the top:\*\*

````python

import json

import sys

````

---

#### 2. \*\*Add these helper functions (anywhere before `main()`):\*\*

````python

def save\_relationships\_snapshot(relationships, filename='relationships\_snapshot.json'):

with open(filename, 'w') as f:

json.dump(relationships, f, indent=2)

print(f"Snapshot saved to {filename}")

def load\_relationships\_snapshot(filename='relationships\_snapshot.json'):

if not os.path.exists(filename):

return None

with open(filename) as f:

return json.load(f)

def compare\_relationships(old, new):

print("\n--- Impact Analysis ---")

for rel\_type in RELATIONSHIP\_KEYS:

old\_set = set(tuple(x) for x in old.get(rel\_type, []))

new\_set = set(tuple(x) for x in new.get(rel\_type, []))

removed = old\_set - new\_set

added = new\_set - old\_set

if removed:

print(f"Removed {rel\_type}: {removed}")

if added:

print(f"Added {rel\_type}: {added}")

print("--- End of Impact Analysis ---\n")

````

---

#### 3. \*\*Modify your `main()` to support snapshot/compare via command-line:\*\*

Replace your `main()` with this version:

````python

def main():

relationships = {k: [] for k in RELATIONSHIP\_KEYS}

class\_names = set()

spring\_info = {}

java\_files = find\_java\_files('test\_samples')

for filename in java\_files:

try:

analyze\_java\_file(filename, relationships, class\_names)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

for filename in java\_files:

try:

analyze\_java\_fields\_and\_associations(filename, relationships, class\_names)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

for filename in java\_files:

try:

analyze\_spring\_annotations(filename, spring\_info)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

for k in relationships:

relationships[k] = sorted(list(set(relationships[k])))

# --- Snapshot/Compare logic ---

if len(sys.argv) > 1 and sys.argv[1] == "snapshot":

save\_relationships\_snapshot(relationships)

return

elif len(sys.argv) > 1 and sys.argv[1] == "compare":

old = load\_relationships\_snapshot()

if old is None:

print("No snapshot found. Run with 'snapshot' first.")

else:

compare\_relationships(old, relationships)

import pprint

pprint.pprint(relationships)

print("\n--- Relationships with variable names ---")

for rel in relationships['composition']:

print(f"{rel[0]} composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['strong\_composition']:

print(f"{rel[0]} strong composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['aggregation']:

print(f"{rel[0]} aggregates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['association']:

print(f"{rel[0]} associates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['inheritance']:

print(f"{rel[0]} inherits {rel[1]}")

for rel in relationships['interface\_implementation']:

print(f"{rel[0]} implements {rel[1]}")

print("\n--- Spring Annotations ---")

for class\_name, anno in spring\_info.get('class\_annotations', []):

print(f"Class '{class\_name}' is annotated with @{anno}")

for var\_name, field\_type, file in spring\_info.get('autowired\_fields', []):

print(f"Field '{var\_name}' of type '{field\_type}' is @Autowired in {file}")

for class\_name, params, file in spring\_info.get('autowired\_ctors', []):

print(f"Constructor of '{class\_name}' is @Autowired with params ({params}) in {file}")

visualize\_relationships(relationships)

````

---

### \*\*How to Use:\*\*

1. \*\*Take a snapshot (before changes):\*\*

```sh

python test\_samples/java\_analyzer.py snapshot

```

2. \*\*Make code changes.\*\*

3. \*\*Compare after changes:\*\*

```sh

python test\_samples/java\_analyzer.py compare

```

You’ll see added/removed relationships as impact analysis!

---

Let me know if you want this as a patch for your file.

yes ... also suggest some code change so can verify

Here’s how to add \*\*snapshot/compare\*\* functionality to your java\_analyzer.py, plus a suggestion for a simple code change to verify the impact analysis.

---

### 1. \*\*Add Imports\*\*

Add these at the top:

````python

import json

import sys

````

---

### 2. \*\*Add Helper Functions\*\*

Place these before `main()`:

````python

def save\_relationships\_snapshot(relationships, filename='relationships\_snapshot.json'):

with open(filename, 'w') as f:

json.dump(relationships, f, indent=2)

print(f"Snapshot saved to {filename}")

def load\_relationships\_snapshot(filename='relationships\_snapshot.json'):

if not os.path.exists(filename):

return None

with open(filename) as f:

return json.load(f)

def compare\_relationships(old, new):

print("\n--- Impact Analysis ---")

for rel\_type in RELATIONSHIP\_KEYS:

old\_set = set(tuple(x) for x in old.get(rel\_type, []))

new\_set = set(tuple(x) for x in new.get(rel\_type, []))

removed = old\_set - new\_set

added = new\_set - old\_set

if removed:

print(f"Removed {rel\_type}: {removed}")

if added:

print(f"Added {rel\_type}: {added}")

print("--- End of Impact Analysis ---\n")

````

---

### 3. \*\*Modify `main()`\*\*

Replace your `main()` with this version:

````python

def main():

relationships = {k: [] for k in RELATIONSHIP\_KEYS}

class\_names = set()

spring\_info = {}

java\_files = find\_java\_files('test\_samples')

for filename in java\_files:

try:

analyze\_java\_file(filename, relationships, class\_names)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

for filename in java\_files:

try:

analyze\_java\_fields\_and\_associations(filename, relationships, class\_names)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

for filename in java\_files:

try:

analyze\_spring\_annotations(filename, spring\_info)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

for k in relationships:

relationships[k] = sorted(list(set(relationships[k])))

# --- Snapshot/Compare logic ---

if len(sys.argv) > 1 and sys.argv[1] == "snapshot":

save\_relationships\_snapshot(relationships)

return

elif len(sys.argv) > 1 and sys.argv[1] == "compare":

old = load\_relationships\_snapshot()

if old is None:

print("No snapshot found. Run with 'snapshot' first.")

else:

compare\_relationships(old, relationships)

import pprint

pprint.pprint(relationships)

print("\n--- Relationships with variable names ---")

for rel in relationships['composition']:

print(f"{rel[0]} composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['strong\_composition']:

print(f"{rel[0]} strong composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['aggregation']:

print(f"{rel[0]} aggregates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['association']:

print(f"{rel[0]} associates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['inheritance']:

print(f"{rel[0]} inherits {rel[1]}")

for rel in relationships['interface\_implementation']:

print(f"{rel[0]} implements {rel[1]}")

print("\n--- Spring Annotations ---")

for class\_name, anno in spring\_info.get('class\_annotations', []):

print(f"Class '{class\_name}' is annotated with @{anno}")

for var\_name, field\_type, file in spring\_info.get('autowired\_fields', []):

print(f"Field '{var\_name}' of type '{field\_type}' is @Autowired in {file}")

for class\_name, params, file in spring\_info.get('autowired\_ctors', []):

print(f"Constructor of '{class\_name}' is @Autowired with params ({params}) in {file}")

visualize\_relationships(relationships)

````

---

### 4. \*\*How to Test\*\*

\*\*A. Take a snapshot (before change):\*\*

```sh

python test\_samples/java\_analyzer.py snapshot

```

\*\*B. Make a code change:\*\*

For example, in one of your Java files (e.g., `UserService.java`), comment out or remove a field like:

```java

// private Order order;

```

or remove an `implements UserApi` from a class declaration.

\*\*C. Run compare (after change):\*\*

```sh

python test\_samples/java\_analyzer.py compare

```

\*\*D. You should see output like:\*\*

```

Removed composition: {('UserService', 'Order', 'order')}

Removed association: {('UserService', 'Order', 'order')}

...

```

---

This will verify your impact analysis is working!

this is output

(v11) PS E:\PYTHON\_PROJECTS\Python-Utilities\class\_viz\_dashboard> python test\_samples/java\_analyzer.py compare

--- Impact Analysis ---

Removed interface\_implementation: {('UserService', 'UserApi')}

--- End of Impact Analysis ---

{'aggregation': [],

'association': [('UserRepository', 'User', 'user'),

('UserService', 'EmailService', 'emailService'),

('UserService', 'Order', 'order'),

('UserService', 'User', 'user'),

('UserService', 'UserRepository', 'userRepository')],

'composition': [('UserService', 'Order', 'order'),

('UserService', 'UserRepository', 'userRepository')],

'inheritance': [('UserRepository', 'BaseRepository'),

('UserService', 'BaseService')],

'interface\_implementation': [],

'strong\_composition': []}

--- Relationships with variable names ---

UserService composes Order via variable 'order'

UserService composes UserRepository via variable 'userRepository'

UserRepository associates User via variable 'user'

UserService associates EmailService via variable 'emailService'

UserService associates Order via variable 'order'

UserService associates User via variable 'user'

UserService associates UserRepository via variable 'userRepository'

UserRepository inherits BaseRepository

UserService inherits BaseService

--- Spring Annotations ---

Class 'EmailService' is annotated with @Service

Class 'UserRepository' is annotated with @Repository

Class 'UserService' is annotated with @Service

Field 'userRepository' of type 'UserRepository' is @Autowired in test\_samples\UserService.java

Constructor of 'UserService' is @Autowired with params (UserRepository userRepository, EmailService emailService) in test\_samples\UserService.java

Diagram saved to class\_diagram.png

(v11) PS E:\PYTHON\_PROJECTS\Python-Utilities\class\_viz\_dashboard>

Your output shows that the \*\*impact analysis is working correctly\*\*!

### What Happened:

- The analyzer detected that the relationship `UserService implements UserApi` was \*\*removed\*\* (likely because you removed `implements UserApi` from the `UserService` class declaration).

- All other relationships (composition, association, inheritance, etc.) are still present and correctly listed.

- The impact analysis output:

```

Removed interface\_implementation: {('UserService', 'UserApi')}

```

This is exactly what you want: it tells you what relationship was lost due to your code change.

---

## ✅ \*\*How to further verify\*\*

- Try removing or commenting out another field or method parameter (e.g., remove `Order order = new Order();` from `processUser`).

- Run `python java\_analyzer.py compare` again.

- You should see a new "Removed" entry in the impact analysis for `composition` or `association`.

---

## \*\*Summary\*\*

- Your snapshot/compare logic is working.

- The analyzer is correctly warning you about relationship changes.

- You can now use this workflow to monitor and warn about architectural changes in your codebase!

(v11) PS E:\PYTHON\_PROJECTS\Python-Utilities\class\_viz\_dashboard> python test\_samples/java\_analyzer.py compare

--- Impact Analysis ---

Removed composition: {('UserService', 'Order', 'order')}

Removed association: {('UserService', 'Order', 'order')}

Removed interface\_implementation: {('UserService', 'UserApi')}

--- End of Impact Analysis ---

{'aggregation': [],

'association': [('UserRepository', 'User', 'user'),

('UserService', 'EmailService', 'emailService'),

('UserService', 'User', 'user'),

('UserService', 'UserRepository', 'userRepository')],

'composition': [('UserService', 'UserRepository', 'userRepository')],

'inheritance': [('UserRepository', 'BaseRepository'),

('UserService', 'BaseService')],

'interface\_implementation': [],

'strong\_composition': []}

--- Relationships with variable names ---

UserService composes UserRepository via variable 'userRepository'

UserRepository associates User via variable 'user'

UserService associates EmailService via variable 'emailService'

UserService associates User via variable 'user'

UserService associates UserRepository via variable 'userRepository'

UserRepository inherits BaseRepository

UserService inherits BaseService

--- Spring Annotations ---

Class 'EmailService' is annotated with @Service

Class 'UserRepository' is annotated with @Repository

Class 'UserService' is annotated with @Service

Field 'userRepository' of type 'UserRepository' is @Autowired in test\_samples\UserService.java

Constructor of 'UserService' is @Autowired with params (UserRepository userRepository, EmailService emailService) in test\_samples\UserService.java

Diagram saved to class\_diagram.png

(v11) PS E:\PYTHON\_PROJECTS\Python-Utilities\class\_viz\_dashboard>

Your \*\*impact analysis is working perfectly\*\*!

### What the Output Shows:

- \*\*Removed composition:\*\* `('UserService', 'Order', 'order')`

- \*\*Removed association:\*\* `('UserService', 'Order', 'order')`

- \*\*Removed interface\_implementation:\*\* `('UserService', 'UserApi')`

This means:

- You removed the `Order order` field (or its usage) from `UserService`, so both the composition and association relationships with `Order` are gone.

- You also removed `implements UserApi` from `UserService`, so the interface implementation relationship is gone.

### The Remaining Relationships:

- All other relationships (with `UserRepository`, `EmailService`, `User`, etc.) are still present and correctly detected.

---

## ✅ \*\*Conclusion\*\*

- The analyzer is accurately detecting and reporting changes in class relationships.

- You can trust this tool to warn you about architectural impacts when code changes are made.

- This workflow is now ready for use in real projects for continuous architecture monitoring!

If you want to test further, try adding/removing other fields, methods, or inheritance in your Java files and rerun the compare—you’ll see the impact immediately.

want a suggestion if some one has commented instead of deleting ..... will our logic catch comment is equivalent to delete ... so we can say you commented then ...impact analysis and when deleted then say ...............you deleted or missing

Great question!

\*\*Yes, your current analyzer logic will treat commented-out code the same as deleted code\*\* for the purposes of relationship detection and impact analysis.

### Why?

- Your analyzer \*\*removes comments\*\* from the source code before running regex analysis:

```python

source = re.sub(r'//.\*?$|/\\*.\*?\\*/', '', source, flags=re.DOTALL | re.MULTILINE)

```

- This means any code that is commented out (using or `/\* ... \*/`) will not be seen by the analyzer.

- As a result, \*\*commented-out fields, methods, or class declarations will be treated as if they are deleted or missing\*\*.

---

## \*\*How to Distinguish "Commented" vs "Deleted"?\*\*

- \*\*Currently:\*\*

The analyzer cannot distinguish between code that is commented out and code that is actually deleted—they both appear as "missing" in the analysis.

- \*\*If you want to differentiate:\*\*

You would need to:

- Parse the code before removing comments.

- Or, use a more advanced parser that can detect commented-out code blocks and report them separately.

---

## \*\*Suggested Message Logic\*\*

- \*\*If a relationship is missing (either by comment or delete):\*\*

- You can say:

\_"The relationship X is missing (possibly due to code being commented out or deleted). Please review the change."\_

- \*\*If you want to be more specific:\*\*

- You would need to enhance your analyzer to check for commented-out code patterns and report them as "commented".

---

## \*\*Summary Table\*\*

| Action | Analyzer Output | Message Suggestion |

|----------------|----------------|--------------------------------------------------------|

| Deleted code | Missing | "You deleted X, which affects these relationships..." |

| Commented code | Missing | "You commented out or deleted X, which affects..." |

---

\*\*In short:\*\*

Your analyzer already treats commented-out code as deleted for impact analysis. If you want to distinguish, you’ll need more advanced parsing. For most teams, treating both as "missing" is sufficient and practical!

perfect .. when finding realtionships ... if there is commented line ...are we skipping those lines to be not counted as realtionship ......otherwise there might be many commented lines which will show wrong picture

Yes, your analyzer \*\*skips commented lines\*\* and does \*\*not\*\* count relationships from commented-out code.

### Why?

- Before extracting relationships, your code runs:

```python

source = re.sub(r'//.\*?$|/\\*.\*?\\*/', '', source, flags=re.DOTALL | re.MULTILINE)

```

- This \*\*removes all single-line (`// ...`) and multi-line (`/\* ... \*/`) comments\*\* from the source code.

- As a result, \*\*any code that is commented out will not be analyzed\*\* and will not appear in the relationships.

---

## \*\*Summary\*\*

- \*\*Commented-out code is ignored\*\* by your analyzer.

- Only active (uncommented) code is used to find relationships.

- This prevents false positives from commented code and ensures your relationship diagram reflects the real, active codebase.

You are safe from the issue of commented lines showing up as relationships!

Certainly! Here’s a \*\*strategy for enterprise-level code relationship analysis as a service\*\*, addressing both priorities \*\*independently and together\*\*, with security and audit in mind:

---

## \*\*Step 1: Codebase Understanding Service (Relationship Explorer)\*\*

\*\*Goal:\*\*

Enable any developer (new, experienced, guest) to quickly understand the structure and relationships in the codebase.

\*\*How to Implement:\*\*

- \*\*Service/API:\*\*

Deploy a microservice that scans the codebase and builds relationship diagrams (class, module, package, etc.).

- \*\*Web Dashboard:\*\*

Provide a UI for interactive exploration of relationships (search, filter, zoom, time-travel).

- \*\*Access Control:\*\*

Integrate with SSO (LDAP/OAuth) to restrict access by user role (guest, dev, lead).

- \*\*Audit:\*\*

Log who accessed what, when, and what was viewed/exported.

- \*\*Reporting:\*\*

Allow export of diagrams/reports for onboarding or documentation.

\*\*Independent Use:\*\*

- Can be used standalone for onboarding, architecture review, or documentation.

---

## \*\*Step 2: Change Impact Analysis Service\*\*

\*\*Goal:\*\*

On every code change, compare new relationships with previous state and report the impact.

\*\*How to Implement:\*\*

- \*\*Service/API:\*\*

Expose endpoints to:

- Upload new code snapshot (or auto-trigger on commit/PR)

- Compare with previous snapshot

- Generate impact report (added/removed/changed relationships)

- \*\*Integration:\*\*

- Integrate with SCM (Bitbucket, GitHub, GitLab) as a pre-merge or pre-push check.

- Optionally, provide CLI/IDE plugin for local use.

- \*\*Notifications:\*\*

Send impact reports to developers/reviewers via email, Slack, or PR comments.

- \*\*Audit:\*\*

Log all analyses, who triggered them, results, and actions taken.

- \*\*Security:\*\*

Only authorized users/services can trigger or view analyses.

\*\*Independent Use:\*\*

- Can be used standalone for code review, CI/CD, or compliance.

---

## \*\*Step 3: Combined Workflow (End-to-End Service)\*\*

\*\*How They Work Together:\*\*

- \*\*On codebase scan:\*\*

Relationship Explorer builds and stores the current state.

- \*\*On code change:\*\*

Impact Analysis Service compares new state to previous, generates report, and updates the explorer.

- \*\*Unified Dashboard:\*\*

Users can view both the current structure and the history of changes/impacts.

- \*\*Audit & Security:\*\*

All actions are logged; access is role-based and monitored.

---

## \*\*Enterprise Features\*\*

- \*\*Authentication/Authorization:\*\*

SSO integration, role-based access (admin, dev, guest, auditor).

- \*\*Audit Logging:\*\*

Every scan, comparison, and access is logged with user, timestamp, and action.

- \*\*Data Retention:\*\*

Store snapshots and reports for compliance and traceability.

- \*\*Observability:\*\*

Metrics, health checks, and error logging (integrate with ELK, Prometheus, etc.).

- \*\*Scalability:\*\*

Stateless services, scalable storage (cloud DB, object storage for diagrams).

- \*\*APIs:\*\*

Well-documented REST/gRPC APIs for integration with other tools.

---

## \*\*Summary Table\*\*

| Service | Purpose | Used Alone? | Used Together? |

|--------------------------|--------------------------------------|-------------|---------------|

| Relationship Explorer | Understand codebase structure | Yes | Yes |

| Impact Analysis Service | Detect/report impact of code changes | Yes | Yes |

| Unified Dashboard/API | Combine both for end-to-end workflow | - | Yes |

---

## \*\*Example User Journeys\*\*

- \*\*New Developer:\*\*

Logs in, explores diagrams, understands architecture, exports docs.

- \*\*Reviewer:\*\*

Receives impact report on PR, reviews before approving merge.

- \*\*Auditor:\*\*

Checks audit logs and historical diagrams for compliance.

---

\*\*This approach gives you modular, secure, auditable, and scalable services for both codebase understanding and change impact—independently or together.\*\*

Absolutely! Here’s a \*\*left-shift, enterprise-level strategy\*\* for code relationship and impact analysis, integrating with developer workflows \*\*before CI/CD\*\* (e.g., at Bitbucket/GitHub check-in), to maximize code quality, transparency, and onboarding:

---

## \*\*1. Left-Shift Integration Points\*\*

- \*\*Pre-commit/Pre-push Hooks:\*\*

Integrate analyzer as a Git hook (pre-commit or pre-push) so developers get instant feedback before code leaves their machine.

- \*\*Pull Request Checks:\*\*

Run analyzer as a required check in Bitbucket/GitHub/GitLab pull requests. Block merges if critical architectural changes are detected without review.

- \*\*IDE Plugins:\*\*

Optional: Provide VS Code/IntelliJ plugin to visualize relationships and impact as developers code.

---

## \*\*2. Developer Workflow\*\*

1. \*\*Developer makes code changes.\*\*

2. \*\*On commit/push or PR creation:\*\*

- Analyzer runs automatically.

- Generates:

- \*\*Current relationship diagram\*\* (before and after change)

- \*\*Impact analysis report\*\* (added/removed/changed relationships)

- \*\*Warns developer\*\* in terminal, IDE, or PR UI about:

- What relationships are impacted

- What classes/files are affected

- Visual diff of diagrams (before/after)

- \*\*Requires developer acknowledgment\*\* or reviewer approval for significant changes.

---

## \*\*3. Reporting & Observability\*\*

- \*\*Store snapshots\*\* of relationships in a central database (with commit hash, author, timestamp).

- \*\*Audit trail\*\* for all changes and analyses.

- \*\*Web dashboard\*\* for:

- Browsing relationship diagrams over time

- Viewing impact reports for any commit/branch

- Searching for dependencies or affected modules

- \*\*Notifications\*\* (Slack, Teams, Email) for high-impact changes.

---

## \*\*4. Reviewer & Onboarding Benefits\*\*

- \*\*Reviewers\*\* see a clear, visual summary of what changed in the codebase’s architecture, not just lines of code.

- \*\*New developers\*\* can:

- Explore the codebase visually

- See how classes interact

- Understand the impact of recent changes

- Search for examples of patterns or anti-patterns

---

## \*\*5. Enterprise Strategy Summary\*\*

- \*\*Automate\*\*: Analyzer runs on every commit, push, and PR.

- \*\*Visualize\*\*: Always generate before/after diagrams and impact reports.

- \*\*Warn\*\*: Block or warn on risky architectural changes.

- \*\*Audit\*\*: Store all results for traceability and compliance.

- \*\*Educate\*\*: Use diagrams and reports for onboarding and documentation.

- \*\*Integrate\*\*: Make analyzer part of the standard dev toolchain (hooks, CI, IDE).

---

## \*\*6. Example Flow\*\*

1. \*\*Dev runs `git commit` or opens PR.\*\*

2. \*\*Analyzer runs:\*\*

- Loads previous snapshot (from DB or last commit)

- Analyzes current state

- Compares and generates:

- Relationship diagram (before/after)

- Impact report (added/removed/changed)

- \*\*Shows results in terminal/PR UI\*\*

- \*\*Blocks push/merge if needed\*\*

3. \*\*Reviewer sees diagrams and impact in PR.\*\*

4. \*\*All data stored for audit and onboarding.\*\*

---

## \*\*7. Tech Stack Suggestions\*\*

- \*\*Analyzer Service:\*\* Python/Java microservice (as you have)

- \*\*Database:\*\* PostgreSQL/MySQL for snapshots and audit

- \*\*Frontend:\*\* React/Angular dashboard

- \*\*CI/CD Integration:\*\* Bitbucket Pipelines, GitHub Actions, GitLab CI

- \*\*IDE Plugins:\*\* VS Code, IntelliJ (optional)

- \*\*Notifications:\*\* Slack, Teams, Email

---

\*\*This approach empowers developers and reviewers, increases codebase transparency, and makes onboarding and code reviews much more effective.\*\*

It’s a true left-shift, “shift-left” DevOps practice for architecture and code quality!

Certainly! Here’s a \*\*strategic outline\*\* for scaling your analyzer from a single-user tool to a robust, enterprise-ready \*\*impact analysis service\*\*:

---

## 1. \*\*Architecture: Service-Oriented & Scalable\*\*

- \*\*Central Service/API:\*\*

Deploy the analyzer as a RESTful microservice (e.g., using FastAPI, Flask, Spring Boot, etc.).

- \*\*Database Backend:\*\*

Use a relational DB (PostgreSQL, MySQL) to store:

- Snapshots of relationships (with timestamps, user, branch, commit hash, etc.)

- Audit logs of all analyses and changes

- User/project metadata

---

## 2. \*\*Integration Points\*\*

- \*\*CI/CD Integration:\*\*

Integrate with GitHub Actions, GitLab CI, Jenkins, or Azure DevOps to trigger analysis on pull requests, merges, or scheduled jobs.

- \*\*IDE Plugins:\*\*

Optional: Provide VS Code/IntelliJ plugins to show impact analysis directly in the developer’s IDE.

- \*\*Web Dashboard:\*\*

Build a dashboard (React, Angular, etc.) for:

- Visualizing class diagrams and relationship changes over time

- Browsing audit logs and reports

- Searching and filtering by user, project, or time

---

## 3. \*\*Audit & Observability\*\*

- \*\*Audit Trail:\*\*

Every analysis run logs:

- Who triggered it

- What files/branches/commits were analyzed

- What relationships changed

- Impact analysis results

- \*\*Observability:\*\*

- Use logging frameworks (ELK, Grafana, Prometheus) for metrics and error tracking.

- Expose health and metrics endpoints for monitoring.

---

## 4. \*\*Reporting & Notifications\*\*

- \*\*Automated Reports:\*\*

- Generate and email (or Slack/Teams) impact analysis reports to relevant teams on every code change.

- Allow users to subscribe to changes in specific modules/classes.

- \*\*Change Summaries:\*\*

- Summarize what relationships were added/removed/modified, and which files/classes are impacted.

---

## 5. \*\*Security & Access Control\*\*

- \*\*Authentication/Authorization:\*\*

- Integrate with corporate SSO (OAuth, LDAP, etc.).

- Control who can trigger analyses, view reports, or access audit logs.

---

## 6. \*\*Extensibility\*\*

- \*\*Plugin System:\*\*

- Allow new language analyzers or custom rules to be plugged in.

- \*\*APIs:\*\*

- Expose APIs for other tools to query relationship data or trigger analyses.

---

## 7. \*\*Example Workflow\*\*

1. \*\*Developer pushes code\*\* → CI triggers analyzer service.

2. \*\*Analyzer runs\*\*, stores relationships, compares with previous snapshot.

3. \*\*Impact report generated\*\* (with links to affected files/classes).

4. \*\*Audit log entry created\*\* (who, what, when, result).

5. \*\*Notification sent\*\* to developer/team.

6. \*\*Dashboard updated\*\* for traceability and review.

---

## 8. \*\*Tech Stack Suggestions\*\*

- \*\*Backend:\*\* Python (FastAPI), Java (Spring Boot), or Node.js

- \*\*Database:\*\* PostgreSQL or MySQL

- \*\*Frontend:\*\* React or Angular

- \*\*CI/CD:\*\* GitHub Actions, Jenkins, etc.

- \*\*Observability:\*\* ELK Stack, Prometheus, Grafana

- \*\*Messaging:\*\* Slack, Teams, Email

---

## 9. \*\*Benefits\*\*

- \*\*Traceability:\*\* Full audit trail of codebase evolution.

- \*\*Proactive Warnings:\*\* Developers see impact before merging.

- \*\*Compliance:\*\* Satisfies audit and change management requirements.

- \*\*Collaboration:\*\* Teams can review and discuss impact reports.

---

\*\*Summary:\*\*

Move from a CLI tool to a service with persistent storage, CI/CD and IDE integration, dashboards, audit logs, and notifications. This enables end-to-end impact analysis, observability, and reporting at scale for enterprise use.

If you want a sample architecture diagram or a more detailed implementation plan, let me know!

great .. want suggestion this looks as single user strategy .... if want to apply as service in corporate arena ....... wht could be best strategy where we keep audit, observability , report ...basically end to end ..where he gets impact analysis ...... think dont reply immediately

give strategy left shift approach before cicd , bitbucket checkin ... we warn , inform developer of wht is happeing and give him complete code analyzer realtionship digram before and after and impact like we are doing from analyzer.py ... strategy at enterprise level ..left shift approach ...... so reviewer is confident of wht going on and new developer know wht is the codebase to study

1. priority - to understand exisitng codebase .. new developer, experience developer , guest ....with realtionships atleast gets wht is in it

2. changes done compare with exisitng code ..... to give impact

now strategise in enterprise how two steps ... each 1 alone and each one idependent also ... as a service with all security, audit

Certainly! Here’s a \*\*strategy for enterprise-level code relationship analysis as a service\*\*, addressing both priorities \*\*independently and together\*\*, with security and audit in mind:

---

## \*\*Step 1: Codebase Understanding Service (Relationship Explorer)\*\*

\*\*Goal:\*\*

Enable any developer (new, experienced, guest) to quickly understand the structure and relationships in the codebase.

\*\*How to Implement:\*\*

- \*\*Service/API:\*\*

Deploy a microservice that scans the codebase and builds relationship diagrams (class, module, package, etc.).

- \*\*Web Dashboard:\*\*

Provide a UI for interactive exploration of relationships (search, filter, zoom, time-travel).

- \*\*Access Control:\*\*

Integrate with SSO (LDAP/OAuth) to restrict access by user role (guest, dev, lead).

- \*\*Audit:\*\*

Log who accessed what, when, and what was viewed/exported.

- \*\*Reporting:\*\*

Allow export of diagrams/reports for onboarding or documentation.

\*\*Independent Use:\*\*

- Can be used standalone for onboarding, architecture review, or documentation.

---

## \*\*Step 2: Change Impact Analysis Service\*\*

\*\*Goal:\*\*

On every code change, compare new relationships with previous state and report the impact.

\*\*How to Implement:\*\*

- \*\*Service/API:\*\*

Expose endpoints to:

- Upload new code snapshot (or auto-trigger on commit/PR)

- Compare with previous snapshot

- Generate impact report (added/removed/changed relationships)

- \*\*Integration:\*\*

- Integrate with SCM (Bitbucket, GitHub, GitLab) as a pre-merge or pre-push check.

- Optionally, provide CLI/IDE plugin for local use.

- \*\*Notifications:\*\*

Send impact reports to developers/reviewers via email, Slack, or PR comments.

- \*\*Audit:\*\*

Log all analyses, who triggered them, results, and actions taken.

- \*\*Security:\*\*

Only authorized users/services can trigger or view analyses.

\*\*Independent Use:\*\*

- Can be used standalone for code review, CI/CD, or compliance.

---

## \*\*Step 3: Combined Workflow (End-to-End Service)\*\*

\*\*How They Work Together:\*\*

- \*\*On codebase scan:\*\*

Relationship Explorer builds and stores the current state.

- \*\*On code change:\*\*

Impact Analysis Service compares new state to previous, generates report, and updates the explorer.

- \*\*Unified Dashboard:\*\*

Users can view both the current structure and the history of changes/impacts.

- \*\*Audit & Security:\*\*

All actions are logged; access is role-based and monitored.

---

## \*\*Enterprise Features\*\*

- \*\*Authentication/Authorization:\*\*

SSO integration, role-based access (admin, dev, guest, auditor).

- \*\*Audit Logging:\*\*

Every scan, comparison, and access is logged with user, timestamp, and action.

- \*\*Data Retention:\*\*

Store snapshots and reports for compliance and traceability.

- \*\*Observability:\*\*

Metrics, health checks, and error logging (integrate with ELK, Prometheus, etc.).

- \*\*Scalability:\*\*

Stateless services, scalable storage (cloud DB, object storage for diagrams).

- \*\*APIs:\*\*

Well-documented REST/gRPC APIs for integration with other tools.

---

## \*\*Summary Table\*\*

| Service | Purpose | Used Alone? | Used Together? |

|--------------------------|--------------------------------------|-------------|---------------|

| Relationship Explorer | Understand codebase structure | Yes | Yes |

| Impact Analysis Service | Detect/report impact of code changes | Yes | Yes |

| Unified Dashboard/API | Combine both for end-to-end workflow | - | Yes |

**Summary Table**

| **Service** | **Purpose** | **Used Alone?** | **Used Together?** |
| --- | --- | --- | --- |
| Relationship Explorer | Understand codebase structure | Yes | Yes |
| Impact Analysis Service | Detect/report impact of code changes | Yes | Yes |
| Unified Dashboard/API | Combine both for end-to-end workflow | - | Yes |

## \*\*Example User Journeys\*\*

- \*\*New Developer:\*\*

Logs in, explores diagrams, understands architecture, exports docs.

- \*\*Reviewer:\*\*

Receives impact report on PR, reviews before approving merge.

- \*\*Auditor:\*\*

Checks audit logs and historical diagrams for compliance.

---

\*\*This approach gives you modular, secure, auditable, and scalable services for both codebase understanding and change impact—independently or together.\*\*

KamleshKG: java\_analyzer.py all files in test\_samples folder to find relationship -- inhettence, association, composition, aggregation , implemnet

Workspace: Collecting workspace informationThe java\_analyzer.py script is designed to analyze all `.java` files in the test\_samples folder and extract class relationships, including:

- \*\*Inheritance\*\* (class extends another class)

- \*\*Association\*\* (uses another class as a parameter, field, or local variable)

- \*\*Composition\*\* (has a field of another class type)

- \*\*Aggregation\*\* (has a collection or plural field of another class type)

- \*\*Implementation\*\* (class implements an interface)

It does this by:

1. Scanning all `.java` files in test\_samples using `find\_java\_files`.

2. Extracting class names, inheritance, and interface implementations with `analyze\_java\_file`.

3. Extracting fields, associations, and compositions with `analyze\_java\_fields\_and\_associations`.

4. Optionally, extracting Spring annotations with `analyze\_spring\_annotations`.

5. Visualizing the relationships with `visualize\_relationships`.

To run the analysis and generate a diagram, simply execute:

```sh

python test\_samples/java\_analyzer.py

```

This will print the relationships and generate a `class\_diagram.png` file showing the relationships between classes in test\_samples.

KamleshKG: output

(v11) PS E:\PYTHON\_PROJECTS\Python-Utilities\class\_viz\_dashboard> python test\_samples/java\_analyzer.py

{'aggregation': [],

'association': [('UserRepository', 'User', 'user'),

('UserService', 'EmailService', 'emailService'),

('UserService', 'Order', 'order'),

('UserService', 'User', 'user'),

('UserService', 'UserRepository', 'userRepository')],

'composition': [('UserService', 'Order', 'order'),

('UserService', 'UserRepository', 'userRepository')],

'inheritance': [('UserRepository', 'BaseRepository'),

('UserService', 'BaseService')],

'interface\_implementation': [('UserService', 'UserApi')],

'strong\_composition': []}

--- Relationships with variable names ---

UserService composes Order via variable 'order'

UserService composes UserRepository via variable 'userRepository'

UserRepository associates User via variable 'user'

UserService associates EmailService via variable 'emailService'

UserService associates Order via variable 'order'

UserService associates User via variable 'user'

UserService associates UserRepository via variable 'userRepository'

UserRepository inherits BaseRepository

UserService inherits BaseService

UserService implements UserApi

--- Spring Annotations ---

Class 'EmailService' is annotated with @Service

Class 'UserRepository' is annotated with @Repository

Class 'UserService' is annotated with @Service

Field 'userRepository' of type 'UserRepository' is @Autowired in test\_samples\UserService.java

Constructor of 'UserService' is @Autowired with params (UserRepository userRepository, EmailService emailService) in test\_samples\UserService.java

Diagram saved to class\_diagram.png

can you verify everthing correct ..are we getting all realtionships also variable which creates realtionship

GitHub Copilot:

KamleshKG: wht is model ... can you guide

GitHub Copilot: In the context of software engineering and programming, \*\*"model"\*\* can have several meanings depending on the context. Here are the most common uses:

---

### 1. \*\*Data Model / Domain Model\*\*

- A \*\*model\*\* often refers to a class or set of classes that represent the data and business logic of your application.

- In Java or Python, a model class typically represents an entity (like `User`, `Order`, `Product`) and contains fields (attributes) and sometimes methods related to that entity.

\*\*Example (Java):\*\*

```java

public class User {

private String name;

private String email;

// getters, setters, etc.

}

```

\*\*Example (Python):\*\*

```python

class User:

def \_\_init\_\_(self, name, email):

self.name = name

self.email = email

```

---

### 2. \*\*MVC Pattern (Model-View-Controller)\*\*

- In the \*\*MVC\*\* architectural pattern, the \*\*Model\*\* is responsible for managing the data, logic, and rules of the application.

- The \*\*View\*\* displays the data, and the \*\*Controller\*\* handles user input.

---

### 3. \*\*Database Model\*\*

- In frameworks like Django (Python) or Spring (Java), a \*\*model\*\* often maps to a database table.

- Each instance of the model represents a row in the table.

---

### 4. \*\*Machine Learning Model\*\*

- In machine learning, a \*\*model\*\* is a mathematical representation trained to make predictions or decisions based on data.

---

## \*\*Summary\*\*

- In your context (Java class diagrams and relationships), a \*\*model\*\* is most likely a class that represents a real-world entity or concept in your application, often used to structure and manage data.

If you have a specific context (e.g., Spring, Django, ML), let me know for a more tailored explanation!

KamleshKG: similar to java\_analyzer.py can you give groovy\_analyzer.py ...

GitHub Copilot: Certainly! Here’s a `groovy\_analyzer.py` script similar in structure to your `java\_analyzer.py`, but adapted for Groovy syntax. It will scan all `.groovy` files in a given directory, extract class relationships (inheritance, association, composition, aggregation, implementation), and visualize them.

````python

import os

import re

from graphviz import Digraph

RELATIONSHIP\_KEYS = [

'aggregation',

'composition',

'strong\_composition',

'association',

'inheritance',

'interface\_implementation'

]

def find\_groovy\_files(root\_dir):

groovy\_files = []

for dirpath, \_, filenames in os.walk(root\_dir):

for fname in filenames:

if fname.endswith('.groovy'):

groovy\_files.append(os.path.join(dirpath, fname))

return groovy\_files

def analyze\_groovy\_file(filepath, relationships, class\_names):

with open(filepath, "r", encoding="utf-8") as f:

source = f.read()

# Remove comments

source = re.sub(r'//.\*?$|/\\*.\*?\\*/', '', source, flags=re.DOTALL | re.MULTILINE)

# Find all class/interface definitions

class\_pattern = re.compile(

r'(class|interface)\s+(\w+)(?:\s+extends\s+(\w+))?(?:\s+implements\s+([\w,\s]+))?', re.MULTILINE)

classes = class\_pattern.findall(source)

for kind, class\_name, parent, interfaces in classes:

class\_names.add(class\_name)

if parent:

relationships['inheritance'].append((class\_name, parent))

if interfaces:

for iface in [i.strip() for i in interfaces.split(',') if i.strip()]:

relationships['interface\_implementation'].append((class\_name, iface))

def analyze\_groovy\_fields\_and\_associations(filepath, relationships, class\_names):

with open(filepath, "r", encoding="utf-8") as f:

source = f.read()

# Remove comments

source = re.sub(r'//.\*?$|/\\*.\*?\\*/', '', source, flags=re.DOTALL | re.MULTILINE)

# Groovy allows def and type inference, but we'll look for explicit types and def

class\_body\_pattern = re.compile(

r'(class|interface)\s+(\w+)[^{]\*\{((?:[^{}]|\{[^{}]\*\})\*)}', re.DOTALL)

for match in class\_body\_pattern.finditer(source):

kind, class\_name, body = match.groups()

# Fields (composition/aggregation)

field\_pattern = re.compile(r'(def|\w+)\s+(\w+)\s\*(=|;)', re.MULTILINE)

for field\_type, var\_name, \_ in field\_pattern.findall(body):

if field\_type in class\_names:

if re.search(r'List<\s\*' + field\_type + r'\s\*>', body) or var\_name.endswith('s'):

relationships['aggregation'].append((class\_name, field\_type, var\_name))

else:

relationships['composition'].append((class\_name, field\_type, var\_name))

# Method parameters (association)

method\_pattern = re.compile(r'(?:def|\w+)\s+\w+\s\*\(([^)]\*)\)', re.MULTILINE)

for params in method\_pattern.findall(body):

for param in params.split(','):

param = param.strip()

if not param:

continue

parts = param.split()

if len(parts) == 2:

param\_type, param\_name = parts

if param\_type in class\_names:

relationships['association'].append((class\_name, param\_type, param\_name))

# Local variables in methods (association)

local\_var\_pattern = re.compile(r'(def|\w+)\s+(\w+)\s\*=\s\*new\s+(\w+)\s\*\(', re.MULTILINE)

for var\_type, var\_name, new\_type in local\_var\_pattern.findall(body):

if new\_type in class\_names:

relationships['association'].append((class\_name, new\_type, var\_name))

def visualize\_relationships(relationships, output\_file='groovy\_class\_diagram'):

dot = Digraph(comment='Groovy Class Diagram', format='png')

dot.attr(rankdir='LR')

# Collect all class names

class\_names = set()

for rels in relationships.values():

for rel in rels:

class\_names.add(rel[0])

class\_names.add(rel[1])

# Add nodes for each class

for cls in class\_names:

dot.node(cls, cls, shape='box')

# Add edges for each relationship type, showing variable name if present

for a, b, var in relationships.get('composition', []):

dot.edge(a, b, label=f"{var} (composes)", arrowhead='diamond', color='green')

for a, b, var in relationships.get('strong\_composition', []):

dot.edge(a, b, label=f"{var} (strong comp.)", arrowhead='diamond', color='red', penwidth='2')

for a, b, var in relationships.get('aggregation', []):

dot.edge(a, b, label=f"{var} (aggregates)", arrowhead='odiamond', color='orange')

for a, b, var in relationships.get('association', []):

dot.edge(a, b, label=f"{var} (associates)", style='dotted', color='purple')

for a, b in relationships.get('inheritance', []):

dot.edge(b, a, label='inherits', arrowhead='onormal', color='black')

for a, b in relationships.get('interface\_implementation', []):

dot.edge(b, a, label='implements', style='dashed', color='blue')

dot.render(output\_file, view=True)

print(f"Diagram saved to {output\_file}.png")

def main():

relationships = {k: [] for k in RELATIONSHIP\_KEYS}

class\_names = set()

groovy\_files = find\_groovy\_files('test\_samples')

# First pass: collect all class names and inheritance/interfaces

for filename in groovy\_files:

try:

analyze\_groovy\_file(filename, relationships, class\_names)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

# Second pass: collect fields and associations

for filename in groovy\_files:

try:

analyze\_groovy\_fields\_and\_associations(filename, relationships, class\_names)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

# Remove duplicates and sort for consistent output

for k in relationships:

relationships[k] = sorted(list(set(relationships[k])))

import pprint

pprint.pprint(relationships)

print("\n--- Relationships with variable names ---")

for rel in relationships['composition']:

print(f"{rel[0]} composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['strong\_composition']:

print(f"{rel[0]} strong composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['aggregation']:

print(f"{rel[0]} aggregates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['association']:

print(f"{rel[0]} associates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['inheritance']:

print(f"{rel[0]} inherits {rel[1]}")

for rel in relationships['interface\_implementation']:

print(f"{rel[0]} implements {rel[1]}")

visualize\_relationships(relationships)

if \_\_name\_\_ == "\_\_main\_\_":

main()

````

\*\*How to use:\*\*

1. Place this script in your project (e.g., `test\_samples/groovy\_analyzer.py`).

2. Make sure you have some `.groovy` files in the test\_samples folder.

3. Run the script:

```sh

python test\_samples/groovy\_analyzer.py

```

4. It will print relationships and generate a `groovy\_class\_diagram.png` file.

\*\*Note:\*\*

Groovy is more dynamic than Java, so static analysis may miss some relationships, especially with `def` and dynamic typing. For best results, use explicit types in your Groovy classes.

KamleshKG: this is output is this ok

(v11) PS E:\PYTHON\_PROJECTS\Python-Utilities\class\_viz\_dashboard> python test\_samples/java\_analyzer.py

{'aggregation': [],

'association': [('UserRepository', 'User', 'user'),

('UserService', 'EmailService', 'emailService'),

('UserService', 'Order', 'order'),

('UserService', 'User', 'user'),

('UserService', 'UserRepository', 'userRepository')],

'composition': [('UserService', 'Order', 'order'),

('UserService', 'UserRepository', 'userRepository')],

'inheritance': [('UserRepository', 'BaseRepository'),

('UserService', 'BaseService')],

'interface\_implementation': [('UserService', 'UserApi')],

'strong\_composition': []}

--- Relationships with variable names ---

UserService composes Order via variable 'order'

UserService composes UserRepository via variable 'userRepository'

UserRepository associates User via variable 'user'

UserService associates EmailService via variable 'emailService'

UserService associates Order via variable 'order'

UserService associates User via variable 'user'

UserService associates UserRepository via variable 'userRepository'

UserRepository inherits BaseRepository

UserService inherits BaseService

UserService implements UserApi

--- Spring Annotations ---

Class 'EmailService' is annotated with @Service

Class 'UserRepository' is annotated with @Repository

Class 'UserService' is annotated with @Service

Field 'userRepository' of type 'UserRepository' is @Autowired in test\_samples\UserService.java

Constructor of 'UserService' is @Autowired with params (UserRepository userRepository, EmailService emailService) in test\_samples\UserService.java

GitHub Copilot: Yes, your output looks correct and matches the expected relationships for a typical Java Spring project. Here’s a quick verification:

- \*\*Inheritance:\*\*

- `UserRepository` inherits `BaseRepository`

- `UserService` inherits `BaseService`

- \*\*Interface Implementation:\*\*

- `UserService` implements `UserApi`

- \*\*Composition:\*\*

- `UserService` composes `Order` via `order`

- `UserService` composes `UserRepository` via `userRepository`

- \*\*Association:\*\*

- `UserRepository` associates `User` via `user`

- `UserService` associates `EmailService`, `Order`, `User`, and `UserRepository` via respective variables

- \*\*Spring Annotations:\*\*

- Detected `@Service`, `@Repository`, and `@Autowired` annotations correctly

- \*\*No aggregation or strong composition\*\* detected, which is fine if your codebase doesn’t use those patterns.

\*\*Conclusion:\*\*

Your analyzer is correctly extracting and displaying class relationships and variable names that create those relationships. The output is as expected for the provided code. If you add more classes or relationships, they should appear in the output as well.

KamleshKG: want your suggestion -- from the java\_analyzer.py if we store in database .... some developer is changing code ...... based on saved realtionship ..... can we suggest him ....due to your changes in this file ...it will effect these all reatlions ....... a kind of warning .... this is sample code only .. imagine we have production code with huge files and logic

GitHub Copilot: This is an excellent idea and a common requirement in large codebases—\*\*impact analysis\*\*. Here’s a high-level approach and suggestions for implementing this:

---

## \*\*How It Could Work\*\*

1. \*\*Store Relationships in a Database\*\*

- After each analysis run, save all extracted relationships (inheritance, association, etc.) in a database table, including the source file and line numbers if possible.

2. \*\*Detect Code Changes\*\*

- Use a version control system (like Git) to detect which files have changed.

- Alternatively, monitor file modification times or use a pre-commit hook.

3. \*\*Compare Old and New Relationships\*\*

- Re-run the analyzer on changed files.

- Compare the new relationships with those stored in the database.

- Identify what relationships have been added, removed, or changed.

4. \*\*Impact Analysis\*\*

- For each changed relationship, find all related classes/files from the database.

- Generate a report or warning:

\_"Changing `UserService.groovy` affects these relationships: [list]. These classes may be impacted: [list]."\_

5. \*\*Notify Developer\*\*

- Show this warning in the IDE, CI/CD pipeline, or as a report.

---

## \*\*Sample Workflow\*\*

1. \*\*Initial Analysis\*\*

- Run analyzer, store relationships in DB:

```

| ClassA | inherits | ClassB | file: ClassA.groovy | line: 5 |

| ... | ... | ... | ... | ... |

```

2. \*\*Developer Changes `ClassA.groovy`\*\*

- Analyzer detects `ClassA` now no longer inherits `ClassB`.

3. \*\*Impact Report\*\*

- System finds all classes that depend on `ClassA`'s inheritance.

- Warns:

\_"Removing inheritance from `ClassA` may break: [ClassC, ClassD] which depend on this relationship."\_

---

## \*\*Benefits\*\*

- \*\*Proactive Warnings:\*\* Developers see the impact before merging code.

- \*\*Better Code Quality:\*\* Reduces accidental breaking changes.

- \*\*Documentation:\*\* Keeps a living map of code relationships.

---

## \*\*Implementation Tips\*\*

- Use a relational database (PostgreSQL, MySQL, SQLite) for storing relationships.

- Store: class name, relationship type, related class, variable name, file, line number, timestamp.

- Use Git hooks or CI/CD to trigger analysis on code changes.

- For large codebases, optimize by analyzing only changed files and their dependencies.

---

## \*\*Summary\*\*

Yes, you can absolutely build a system that warns developers about the impact of their changes by storing and comparing class relationships. This is a valuable tool for large, complex projects and can be integrated into your development workflow for better maintainability and safety.2. \*\*Developer Changes `ClassA.groovy`\*\*

- Analyzer detects `ClassA` now no longer inherits `ClassB`.

3. \*\*Impact Report\*\*

- System finds all classes that depend on `ClassA`'s inheritance.

- Warns:

\_"Removing inheritance from `ClassA` may break: [ClassC, ClassD] which depend on this relationship."\_

---

## \*\*Benefits\*\*

- \*\*Proactive Warnings:\*\* Developers see the impact before merging code.

- \*\*Better Code Quality:\*\* Reduces accidental breaking changes.

- \*\*Documentation:\*\* Keeps a living map of code relationships.

---

## \*\*Implementation Tips\*\*

- Use a relational database (PostgreSQL, MySQL, SQLite) for storing relationships.

- Store: class name, relationship type, related class, variable name, file, line number, timestamp.

- Use Git hooks or CI/CD to trigger analysis on code changes.

- For large codebases, optimize by analyzing only changed files and their dependencies.

---

## \*\*Summary\*\*

Yes, you can absolutely build a system that warns developers about the impact of their changes by storing and comparing class relationships. This is a valuable tool for large, complex projects and can be integrated into your development workflow for better maintainability and safety.

KamleshKG: can we create such model uding java\_analyzer.py .....impact analysis .. and test localy

GitHub Copilot: Yes, you \*\*can\*\* create a model for impact analysis using your java\_analyzer.py! Here’s how you can approach it and test locally:

---

## \*\*1. Store Relationships as a Model\*\*

After running your analyzer, save the relationships (e.g., as a JSON file or in a SQLite database).

Example for JSON:

```python

import json

# After relationships are built

with open('relationships\_snapshot.json', 'w') as f:

json.dump(relationships, f, indent=2)

```

---

## \*\*2. Detect Code Changes\*\*

- Use file modification times, or

- Use `git diff` to see which files have changed.

---

## \*\*3. Re-analyze and Compare\*\*

After a code change:

- Run java\_analyzer.py again to get the new relationships.

- Load the previous snapshot and compare.

Example comparison logic:

```python

with open('relationships\_snapshot.json') as f:

old\_relationships = json.load(f)

# Assume `relationships` is the new one

for rel\_type in RELATIONSHIP\_KEYS:

removed = set(old\_relationships[rel\_type]) - set(relationships[rel\_type])

added = set(relationships[rel\_type]) - set(old\_relationships[rel\_type])

if removed:

print(f"Removed {rel\_type}: {removed}")

if added:

print(f"Added {rel\_type}: {added}")

```

---

## \*\*4. Impact Analysis Output\*\*

For each removed or added relationship, print or log a warning:

- “Changing `UserService.java` removed inheritance from `BaseService`. This may affect: [list classes that depended on this].”

---

## \*\*5. Test Locally\*\*

1. Run java\_analyzer.py and save the snapshot.

2. Make a code change.

3. Run java\_analyzer.py again and compare.

4. See the printed warnings about impacted relationships.

---

## \*\*Summary\*\*

- \*\*Yes, you can do this locally.\*\*

- Use your analyzer to snapshot relationships.

- After changes, compare snapshots and print impact warnings.

- You can use JSON or SQLite for storage.

If you want, I can provide a ready-to-use code snippet for the snapshot/compare logic!

KamleshKG: pls. provide a ready-to-use code smippet for the snapshot/compare

GitHub Copilot: Certainly! Here’s a \*\*ready-to-use code snippet\*\* you can add to your java\_analyzer.py for snapshotting and comparing relationships for impact analysis.

---

#### 1. \*\*Add these imports at the top:\*\*

````python

import json

import sys

````

---

#### 2. \*\*Add these helper functions (anywhere before `main()`):\*\*

````python

def save\_relationships\_snapshot(relationships, filename='relationships\_snapshot.json'):

with open(filename, 'w') as f:

json.dump(relationships, f, indent=2)

print(f"Snapshot saved to {filename}")

def load\_relationships\_snapshot(filename='relationships\_snapshot.json'):

if not os.path.exists(filename):

return None

with open(filename) as f:

return json.load(f)

def compare\_relationships(old, new):

print("\n--- Impact Analysis ---")

for rel\_type in RELATIONSHIP\_KEYS:

old\_set = set(tuple(x) for x in old.get(rel\_type, []))

new\_set = set(tuple(x) for x in new.get(rel\_type, []))

removed = old\_set - new\_set

added = new\_set - old\_set

if removed:

print(f"Removed {rel\_type}: {removed}")

if added:

print(f"Added {rel\_type}: {added}")

print("--- End of Impact Analysis ---\n")

````

---

#### 3. \*\*Modify your `main()` to support snapshot/compare via command-line:\*\*

Replace your `main()` with this version:

````python

def main():

relationships = {k: [] for k in RELATIONSHIP\_KEYS}

class\_names = set()

spring\_info = {}

java\_files = find\_java\_files('test\_samples')

for filename in java\_files:

try:

analyze\_java\_file(filename, relationships, class\_names)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

for filename in java\_files:

try:

analyze\_java\_fields\_and\_associations(filename, relationships, class\_names)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

for filename in java\_files:

try:

analyze\_spring\_annotations(filename, spring\_info)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

for k in relationships:

relationships[k] = sorted(list(set(relationships[k])))

# --- Snapshot/Compare logic ---

if len(sys.argv) > 1 and sys.argv[1] == "snapshot":

save\_relationships\_snapshot(relationships)

return

elif len(sys.argv) > 1 and sys.argv[1] == "compare":

old = load\_relationships\_snapshot()

if old is None:

print("No snapshot found. Run with 'snapshot' first.")

else:

compare\_relationships(old, relationships)

import pprint

pprint.pprint(relationships)

print("\n--- Relationships with variable names ---")

for rel in relationships['composition']:

print(f"{rel[0]} composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['strong\_composition']:

print(f"{rel[0]} strong composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['aggregation']:

print(f"{rel[0]} aggregates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['association']:

print(f"{rel[0]} associates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['inheritance']:

print(f"{rel[0]} inherits {rel[1]}")

for rel in relationships['interface\_implementation']:

print(f"{rel[0]} implements {rel[1]}")

print("\n--- Spring Annotations ---")

for class\_name, anno in spring\_info.get('class\_annotations', []):

print(f"Class '{class\_name}' is annotated with @{anno}")

for var\_name, field\_type, file in spring\_info.get('autowired\_fields', []):

print(f"Field '{var\_name}' of type '{field\_type}' is @Autowired in {file}")

for class\_name, params, file in spring\_info.get('autowired\_ctors', []):

print(f"Constructor of '{class\_name}' is @Autowired with params ({params}) in {file}")

visualize\_relationships(relationships)

````

---

### \*\*How to Use:\*\*

1. \*\*Take a snapshot (before changes):\*\*

```sh

python test\_samples/java\_analyzer.py snapshot

```

2. \*\*Make code changes.\*\*

3. \*\*Compare after changes:\*\*

```sh

python test\_samples/java\_analyzer.py compare

```

You’ll see added/removed relationships as impact analysis!

---

Let me know if you want this as a patch for your file.

KamleshKG: yes ... also suggest some code change so can verify

GitHub Copilot: Here’s how to add \*\*snapshot/compare\*\* functionality to your java\_analyzer.py, plus a suggestion for a simple code change to verify the impact analysis.

---

### 1. \*\*Add Imports\*\*

Add these at the top:

````python

import json

import sys

````

---

### 2. \*\*Add Helper Functions\*\*

Place these before `main()`:

````python

def save\_relationships\_snapshot(relationships, filename='relationships\_snapshot.json'):

with open(filename, 'w') as f:

json.dump(relationships, f, indent=2)

print(f"Snapshot saved to {filename}")

def load\_relationships\_snapshot(filename='relationships\_snapshot.json'):

if not os.path.exists(filename):

return None

with open(filename) as f:

return json.load(f)

def compare\_relationships(old, new):

print("\n--- Impact Analysis ---")

for rel\_type in RELATIONSHIP\_KEYS:

old\_set = set(tuple(x) for x in old.get(rel\_type, []))

new\_set = set(tuple(x) for x in new.get(rel\_type, []))

removed = old\_set - new\_set

added = new\_set - old\_set

if removed:

print(f"Removed {rel\_type}: {removed}")

if added:

print(f"Added {rel\_type}: {added}")

print("--- End of Impact Analysis ---\n")

````

---

### 3. \*\*Modify `main()`\*\*

Replace your `main()` with this version:

````python

def main():

relationships = {k: [] for k in RELATIONSHIP\_KEYS}

class\_names = set()

spring\_info = {}

java\_files = find\_java\_files('test\_samples')

for filename in java\_files:

try:

analyze\_java\_file(filename, relationships, class\_names)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

for filename in java\_files:

try:

analyze\_java\_fields\_and\_associations(filename, relationships, class\_names)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

for filename in java\_files:

try:

analyze\_spring\_annotations(filename, spring\_info)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

for k in relationships:

relationships[k] = sorted(list(set(relationships[k])))

# --- Snapshot/Compare logic ---

if len(sys.argv) > 1 and sys.argv[1] == "snapshot":

save\_relationships\_snapshot(relationships)

return

elif len(sys.argv) > 1 and sys.argv[1] == "compare":

old = load\_relationships\_snapshot()

if old is None:

print("No snapshot found. Run with 'snapshot' first.")

else:

compare\_relationships(old, relationships)

import pprint

pprint.pprint(relationships)

print("\n--- Relationships with variable names ---")

for rel in relationships['composition']:

print(f"{rel[0]} composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['strong\_composition']:

print(f"{rel[0]} strong composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['aggregation']:

print(f"{rel[0]} aggregates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['association']:

print(f"{rel[0]} associates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['inheritance']:

print(f"{rel[0]} inherits {rel[1]}")

for rel in relationships['interface\_implementation']:

print(f"{rel[0]} implements {rel[1]}")

print("\n--- Spring Annotations ---")

for class\_name, anno in spring\_info.get('class\_annotations', []):

print(f"Class '{class\_name}' is annotated with @{anno}")

for var\_name, field\_type, file in spring\_info.get('autowired\_fields', []):

print(f"Field '{var\_name}' of type '{field\_type}' is @Autowired in {file}")

for class\_name, params, file in spring\_info.get('autowired\_ctors', []):

print(f"Constructor of '{class\_name}' is @Autowired with params ({params}) in {file}")

visualize\_relationships(relationships)

````

---

### 4. \*\*How to Test\*\*

\*\*A. Take a snapshot (before change):\*\*

```sh

python test\_samples/java\_analyzer.py snapshot

```

\*\*B. Make a code change:\*\*

For example, in one of your Java files (e.g., `UserService.java`), comment out or remove a field like:

```java

// private Order order;

```

or remove an `implements UserApi` from a class declaration.

\*\*C. Run compare (after change):\*\*

```sh

python test\_samples/java\_analyzer.py compare

```

\*\*D. You should see output like:\*\*

```

Removed composition: {('UserService', 'Order', 'order')}

Removed association: {('UserService', 'Order', 'order')}

...

```

---

This will verify your impact analysis is working!

KamleshKG: this is output

(v11) PS E:\PYTHON\_PROJECTS\Python-Utilities\class\_viz\_dashboard> python test\_samples/java\_analyzer.py compare

--- Impact Analysis ---

Removed interface\_implementation: {('UserService', 'UserApi')}

--- End of Impact Analysis ---

{'aggregation': [],

'association': [('UserRepository', 'User', 'user'),

('UserService', 'EmailService', 'emailService'),

('UserService', 'Order', 'order'),

('UserService', 'User', 'user'),

('UserService', 'UserRepository', 'userRepository')],

'composition': [('UserService', 'Order', 'order'),

('UserService', 'UserRepository', 'userRepository')],

'inheritance': [('UserRepository', 'BaseRepository'),

('UserService', 'BaseService')],

'interface\_implementation': [],

'strong\_composition': []}

--- Relationships with variable names ---

UserService composes Order via variable 'order'

UserService composes UserRepository via variable 'userRepository'

UserRepository associates User via variable 'user'

UserService associates EmailService via variable 'emailService'

UserService associates Order via variable 'order'

UserService associates User via variable 'user'

UserService associates UserRepository via variable 'userRepository'

UserRepository inherits BaseRepository

UserService inherits BaseService

--- Spring Annotations ---

Class 'EmailService' is annotated with @Service

Class 'UserRepository' is annotated with @Repository

Class 'UserService' is annotated with @Service

Field 'userRepository' of type 'UserRepository' is @Autowired in test\_samples\UserService.java

Constructor of 'UserService' is @Autowired with params (UserRepository userRepository, EmailService emailService) in test\_samples\UserService.java

Diagram saved to class\_diagram.png

(v11) PS E:\PYTHON\_PROJECTS\Python-Utilities\class\_viz\_dashboard>

GitHub Copilot: Your output shows that the \*\*impact analysis is working correctly\*\*!

### What Happened:

- The analyzer detected that the relationship `UserService implements UserApi` was \*\*removed\*\* (likely because you removed `implements UserApi` from the `UserService` class declaration).

- All other relationships (composition, association, inheritance, etc.) are still present and correctly listed.

- The impact analysis output:

```

Removed interface\_implementation: {('UserService', 'UserApi')}

```

This is exactly what you want: it tells you what relationship was lost due to your code change.

---

## ✅ \*\*How to further verify\*\*

- Try removing or commenting out another field or method parameter (e.g., remove `Order order = new Order();` from `processUser`).

- Run `python java\_analyzer.py compare` again.

- You should see a new "Removed" entry in the impact analysis for `composition` or `association`.

---

## \*\*Summary\*\*

- Your snapshot/compare logic is working.

- The analyzer is correctly warning you about relationship changes.

- You can now use this workflow to monitor and warn about architectural changes in your codebase!

KamleshKG: (v11) PS E:\PYTHON\_PROJECTS\Python-Utilities\class\_viz\_dashboard> python test\_samples/java\_analyzer.py compare

--- Impact Analysis ---

Removed composition: {('UserService', 'Order', 'order')}

Removed association: {('UserService', 'Order', 'order')}

Removed interface\_implementation: {('UserService', 'UserApi')}

--- End of Impact Analysis ---

{'aggregation': [],

'association': [('UserRepository', 'User', 'user'),

('UserService', 'EmailService', 'emailService'),

('UserService', 'User', 'user'),

('UserService', 'UserRepository', 'userRepository')],

'composition': [('UserService', 'UserRepository', 'userRepository')],

'inheritance': [('UserRepository', 'BaseRepository'),

('UserService', 'BaseService')],

'interface\_implementation': [],

'strong\_composition': []}

--- Relationships with variable names ---

UserService composes UserRepository via variable 'userRepository'

UserRepository associates User via variable 'user'

UserService associates EmailService via variable 'emailService'

UserService associates User via variable 'user'

UserService associates UserRepository via variable 'userRepository'

UserRepository inherits BaseRepository

UserService inherits BaseService

--- Spring Annotations ---

Class 'EmailService' is annotated with @Service

Class 'UserRepository' is annotated with @Repository

Class 'UserService' is annotated with @Service

Field 'userRepository' of type 'UserRepository' is @Autowired in test\_samples\UserService.java

Constructor of 'UserService' is @Autowired with params (UserRepository userRepository, EmailService emailService) in test\_samples\UserService.java

Diagram saved to class\_diagram.png

(v11) PS E:\PYTHON\_PROJECTS\Python-Utilities\class\_viz\_dashboard>

GitHub Copilot: Your \*\*impact analysis is working perfectly\*\*!

### What the Output Shows:

- \*\*Removed composition:\*\* `('UserService', 'Order', 'order')`

- \*\*Removed association:\*\* `('UserService', 'Order', 'order')`

- \*\*Removed interface\_implementation:\*\* `('UserService', 'UserApi')`

This means:

- You removed the `Order order` field (or its usage) from `UserService`, so both the composition and association relationships with `Order` are gone.

- You also removed `implements UserApi` from `UserService`, so the interface implementation relationship is gone.

### The Remaining Relationships:

- All other relationships (with `UserRepository`, `EmailService`, `User`, etc.) are still present and correctly detected.

---

## ✅ \*\*Conclusion\*\*

- The analyzer is accurately detecting and reporting changes in class relationships.

- You can trust this tool to warn you about architectural impacts when code changes are made.

- This workflow is now ready for use in real projects for continuous architecture monitoring!

If you want to test further, try adding/removing other fields, methods, or inheritance in your Java files and rerun the compare—you’ll see the impact immediately.

KamleshKG: want a suggestion if some one has commented instead of deleting ..... will our logic catch comment is equivalent to delete ... so we can say you commented then ...impact analysis and when deleted then say ...............you deleted or missing

GitHub Copilot: Great question!

\*\*Yes, your current analyzer logic will treat commented-out code the same as deleted code\*\* for the purposes of relationship detection and impact analysis.

### Why?

- Your analyzer \*\*removes comments\*\* from the source code before running regex analysis:

```python

source = re.sub(r'//.\*?$|/\\*.\*?\\*/', '', source, flags=re.DOTALL | re.MULTILINE)

```

- This means any code that is commented out (using or `/\* ... \*/`) will not be seen by the analyzer.

- As a result, \*\*commented-out fields, methods, or class declarations will be treated as if they are deleted or missing\*\*.

---

## \*\*How to Distinguish "Commented" vs "Deleted"?\*\*

- \*\*Currently:\*\*

The analyzer cannot distinguish between code that is commented out and code that is actually deleted—they both appear as "missing" in the analysis.

- \*\*If you want to differentiate:\*\*

You would need to:

- Parse the code before removing comments.

- Or, use a more advanced parser that can detect commented-out code blocks and report them separately.

---

## \*\*Suggested Message Logic\*\*

- \*\*If a relationship is missing (either by comment or delete):\*\*

- You can say:

\_"The relationship X is missing (possibly due to code being commented out or deleted). Please review the change."\_

- \*\*If you want to be more specific:\*\*

- You would need to enhance your analyzer to check for commented-out code patterns and report them as "commented".

---

## \*\*Summary Table\*\*

| Action | Analyzer Output | Message Suggestion |

|----------------|----------------|--------------------------------------------------------|

| Deleted code | Missing | "You deleted X, which affects these relationships..." |

| Commented code | Missing | "You commented out or deleted X, which affects..." |

---

\*\*In short:\*\*

Your analyzer already treats commented-out code as deleted for impact analysis. If you want to distinguish, you’ll need more advanced parsing. For most teams, treating both as "missing" is sufficient and practical!

KamleshKG: perfect .. when finding realtionships ... if there is commented line ...are we skipping those lines to be not counted as realtionship ......otherwise there might be many commented lines which will show wrong picture

GitHub Copilot: Yes, your analyzer \*\*skips commented lines\*\* and does \*\*not\*\* count relationships from commented-out code.

### Why?

- Before extracting relationships, your code runs:

```python

source = re.sub(r'//.\*?$|/\\*.\*?\\*/', '', source, flags=re.DOTALL | re.MULTILINE)

```

- This \*\*removes all single-line (`// ...`) and multi-line (`/\* ... \*/`) comments\*\* from the source code.

- As a result, \*\*any code that is commented out will not be analyzed\*\* and will not appear in the relationships.

---

## \*\*Summary\*\*

- \*\*Commented-out code is ignored\*\* by your analyzer.

- Only active (uncommented) code is used to find relationships.

- This prevents false positives from commented code and ensures your relationship diagram reflects the real, active codebase.

You are safe from the issue of commented lines showing up as relationships!

KamleshKG: perfect .....you are champ

GitHub Copilot:

KamleshKG: i have opened analyzer.py , groovy\_analyzer.py can you verify both commented lines we are not taking as realtionhip ...also we want to implete impact logic in both files .....similar to we did in java\_analyzer.py

GitHub Copilot: ### 1. \*\*Commented Lines Handling\*\*

#### \*\*groovy\_analyzer.py\*\*

- ✅ \*\*Already correct:\*\*

You use:

```python

source = re.sub(r'//.\*?$|/\\*.\*?\\*/', '', source, flags=re.DOTALL | re.MULTILINE)

```

This removes all comments before relationship extraction.

\*\*Commented-out code is ignored and not counted as a relationship.\*\*

#### \*\*analyzer.py\*\* (Python analyzer)

- ❌ \*\*Not present:\*\*

The code does \*\*not\*\* currently remove comments before parsing.

However, Python's `ast` module ignores comments by default, so commented-out code is not parsed as AST nodes and thus not counted as a relationship.

\*\*So, commented-out code is also ignored.\*\*

---

### 2. \*\*Impact Analysis Logic\*\*

You want to add snapshot/compare logic (like in `java\_analyzer.py`) to both files.

---

## \*\*Ready-to-use Patch for Both Files\*\*

### \*\*A. Add to Top of Each File\*\*

```python

import json

import sys

```

---

### \*\*B. Add Helper Functions (before `main()`)\*\*

```python

def save\_relationships\_snapshot(relationships, filename='relationships\_snapshot.json'):

with open(filename, 'w') as f:

json.dump(relationships, f, indent=2)

print(f"Snapshot saved to {filename}")

def load\_relationships\_snapshot(filename='relationships\_snapshot.json'):

if not os.path.exists(filename):

return None

with open(filename) as f:

return json.load(f)

def compare\_relationships(old, new):

print("\n--- Impact Analysis ---")

for rel\_type in RELATIONSHIP\_KEYS:

old\_set = set(tuple(x) for x in old.get(rel\_type, []))

new\_set = set(tuple(x) for x in new.get(rel\_type, []))

removed = old\_set - new\_set

added = new\_set - old\_set

if removed:

print(f"Removed {rel\_type}: {removed}")

if added:

print(f"Added {rel\_type}: {added}")

print("--- End of Impact Analysis ---\n")

```

---

### \*\*C. Modify `main()` in Each File\*\*

Replace the start of your `main()` with:

```python

def main():

relationships = {k: [] for k in RELATIONSHIP\_KEYS}

# ...existing code to analyze files and fill relationships...

# Remove duplicates and sort for consistent output

for k in relationships:

relationships[k] = sorted(list(set(relationships[k])))

# --- Snapshot/Compare logic ---

if len(sys.argv) > 1 and sys.argv[1] == "snapshot":

save\_relationships\_snapshot(relationships)

return

elif len(sys.argv) > 1 and sys.argv[1] == "compare":

old = load\_relationships\_snapshot()

if old is None:

print("No snapshot found. Run with 'snapshot' first.")

else:

compare\_relationships(old, relationships)

```

Then continue with your pretty-printing and visualization code.

---

## \*\*Summary\*\*

- \*\*Commented code is ignored in both analyzers.\*\*

- \*\*Add the above snapshot/compare logic\*\* to both files for impact analysis.

- \*\*Usage:\*\*

- Take a snapshot:

`python analyzer.py snapshot`

- Make code changes.

- Compare:

`python analyzer.py compare`

Let me know if you want the full patch for either file!

KamleshKG: yes give full patch for both individual files

GitHub Copilot: Certainly! Here are \*\*full patches\*\* for both analyzer.py (Python) and `groovy\_analyzer.py` (Groovy) to add \*\*impact analysis\*\* (snapshot/compare) logic, following the same pattern as your Java analyzer.

---

## 1. Patch for analyzer.py (Python analyzer)

Replace your entire file with the following:

````python

import ast

import os

import json

import sys

from graphviz import Digraph

RELATIONSHIP\_KEYS = [

'aggregation',

'composition',

'strong\_composition',

'association',

'inheritance',

'interface\_implementation'

]

def find\_py\_files(root\_dir):

py\_files = []

for dirpath, \_, filenames in os.walk(root\_dir):

for fname in filenames:

if fname.endswith('.py'):

py\_files.append(os.path.join(dirpath, fname))

return py\_files

def analyze\_file(filepath, relationships):

with open(filepath, "r", encoding="utf-8") as f:

source = f.read()

tree = ast.parse(source, filename=filepath)

class\_defs = {node.name: node for node in tree.body if isinstance(node, ast.ClassDef)}

class\_names = set(class\_defs.keys())

# Inheritance

for class\_name, class\_node in class\_defs.items():

for base in class\_node.bases:

if isinstance(base, ast.Name):

relationships['inheritance'].append((class\_name, base.id))

elif isinstance(base, ast.Attribute):

relationships['inheritance'].append((class\_name, base.attr))

# Composition, strong composition, aggregation, interface implementation

for class\_name, class\_node in class\_defs.items():

base\_names = [b.id if isinstance(b, ast.Name) else getattr(b, 'attr', None) for b in class\_node.bases]

if 'Drivable' in base\_names:

relationships['interface\_implementation'].append((class\_name, 'Drivable'))

for stmt in class\_node.body:

# Detect composition/strong composition via type annotation or assignment in \_\_init\_\_

if isinstance(stmt, ast.FunctionDef) and stmt.name == "\_\_init\_\_":

for node in ast.walk(stmt):

# Handle annotated assignments (AnnAssign)

if isinstance(node, ast.AnnAssign):

if (isinstance(node.target, ast.Attribute) and

isinstance(node.target.value, ast.Name) and

node.target.value.id == "self"):

var\_name = node.target.attr

# List[Type]

if isinstance(node.annotation, ast.Subscript) and getattr(node.annotation.value, 'id', None) == "List":

elt\_type = None

sub = node.annotation.slice

if isinstance(sub, ast.Name):

elt\_type = sub.id

elif hasattr(sub, 'id'):

elt\_type = sub.id

if elt\_type and elt\_type in class\_names:

if var\_name.startswith("\_\_"):

relationships['strong\_composition'].append((class\_name, elt\_type, var\_name))

relationships['composition'].append((class\_name, elt\_type, var\_name))

else:

relationships['composition'].append((class\_name, elt\_type, var\_name))

# Direct type annotation (e.g. Engine)

elif isinstance(node.annotation, ast.Name):

elt\_type = node.annotation.id

if elt\_type and elt\_type in class\_names:

if var\_name.startswith("\_\_"):

relationships['strong\_composition'].append((class\_name, elt\_type, var\_name))

relationships['composition'].append((class\_name, elt\_type, var\_name))

else:

relationships['composition'].append((class\_name, elt\_type, var\_name))

# Handle assignments (Assign)

if isinstance(node, ast.Assign):

for target in node.targets:

if (isinstance(target, ast.Attribute) and

isinstance(target.value, ast.Name) and

target.value.id == "self" and

isinstance(node.value, ast.Call) and

isinstance(node.value.func, ast.Name)):

attr\_name = target.attr

type\_name = node.value.func.id

if type\_name in class\_names:

if attr\_name.startswith("\_\_"):

relationships['strong\_composition'].append((class\_name, type\_name, attr\_name))

relationships['composition'].append((class\_name, type\_name, attr\_name))

else:

relationships['composition'].append((class\_name, type\_name, attr\_name))

# For dataclass fields (aggregation)

if isinstance(stmt, ast.AnnAssign):

if (isinstance(stmt.annotation, ast.Subscript) and

getattr(stmt.annotation.value, 'id', None) == "List"):

elt\_type = None

sub = stmt.annotation.slice

if isinstance(sub, ast.Name):

elt\_type = sub.id

elif hasattr(sub, 'id'):

elt\_type = sub.id

var\_name = stmt.target.id if isinstance(stmt.target, ast.Name) else None

if elt\_type and var\_name and elt\_type in class\_names:

relationships['aggregation'].append((class\_name, elt\_type, var\_name))

# Association: method parameters or local variables using other classes

for class\_name, class\_node in class\_defs.items():

for stmt in class\_node.body:

if isinstance(stmt, ast.FunctionDef):

# Parameters

for arg in stmt.args.args:

if arg.annotation and isinstance(arg.annotation, ast.Name):

param\_type = arg.annotation.id

if param\_type in class\_names and param\_type != class\_name:

relationships['association'].append((class\_name, param\_type, arg.arg))

# Local variables

for node in ast.walk(stmt):

if isinstance(node, ast.Assign):

if isinstance(node.value, ast.Call) and isinstance(node.value.func, ast.Name):

type\_name = node.value.func.id

if type\_name in class\_names and type\_name != class\_name:

# Get variable name

var\_name = node.targets[0].id if isinstance(node.targets[0], ast.Name) else None

if var\_name:

relationships['association'].append((class\_name, type\_name, var\_name))

def visualize\_relationships(relationships, output\_file='class\_diagram'):

dot = Digraph(comment='Class Diagram', format='png')

dot.attr(rankdir='LR')

# Collect all class names

class\_names = set()

for rels in relationships.values():

for rel in rels:

class\_names.add(rel[0])

class\_names.add(rel[1])

# Add nodes for each class

for cls in class\_names:

dot.node(cls, cls, shape='box')

# Add edges for each relationship type, showing variable name if present

for a, b, var in relationships.get('composition', []):

dot.edge(a, b, label=f"{var} (composes)", arrowhead='diamond', color='green')

for a, b, var in relationships.get('strong\_composition', []):

dot.edge(a, b, label=f"{var} (strong comp.)", arrowhead='diamond', color='red', penwidth='2')

for a, b, var in relationships.get('aggregation', []):

dot.edge(a, b, label=f"{var} (aggregates)", arrowhead='odiamond', color='orange')

for a, b, var in relationships.get('association', []):

dot.edge(a, b, label=f"{var} (associates)", style='dotted', color='purple')

for a, b in relationships.get('inheritance', []):

dot.edge(b, a, label='inherits', arrowhead='onormal', color='black')

for a, b in relationships.get('interface\_implementation', []):

dot.edge(b, a, label='implements', style='dashed', color='blue')

dot.render(output\_file, view=True)

print(f"Diagram saved to {output\_file}.png")

# --- Impact Analysis Helpers ---

def save\_relationships\_snapshot(relationships, filename='relationships\_snapshot.json'):

with open(filename, 'w') as f:

json.dump(relationships, f, indent=2)

print(f"Snapshot saved to {filename}")

def load\_relationships\_snapshot(filename='relationships\_snapshot.json'):

if not os.path.exists(filename):

return None

with open(filename) as f:

return json.load(f)

def compare\_relationships(old, new):

print("\n--- Impact Analysis ---")

for rel\_type in RELATIONSHIP\_KEYS:

old\_set = set(tuple(x) for x in old.get(rel\_type, []))

new\_set = set(tuple(x) for x in new.get(rel\_type, []))

removed = old\_set - new\_set

added = new\_set - old\_set

if removed:

print(f"Removed {rel\_type}: {removed}")

if added:

print(f"Added {rel\_type}: {added}")

print("--- End of Impact Analysis ---\n")

def main():

relationships = {k: [] for k in RELATIONSHIP\_KEYS}

py\_files = find\_py\_files('test\_samples')

for filename in py\_files:

try:

analyze\_file(filename, relationships)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

# Remove duplicates and sort for consistent output

for k in relationships:

relationships[k] = sorted(list(set(relationships[k])))

# --- Snapshot/Compare logic ---

if len(sys.argv) > 1 and sys.argv[1] == "snapshot":

save\_relationships\_snapshot(relationships)

return

elif len(sys.argv) > 1 and sys.argv[1] == "compare":

old = load\_relationships\_snapshot()

if old is None:

print("No snapshot found. Run with 'snapshot' first.")

else:

compare\_relationships(old, relationships)

import pprint

pprint.pprint(relationships)

# Print relationships with variable names

print("\n--- Relationships with variable names ---")

for rel in relationships['composition']:

print(f"{rel[0]} composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['strong\_composition']:

print(f"{rel[0]} strong composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['aggregation']:

print(f"{rel[0]} aggregates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['association']:

print(f"{rel[0]} associates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['inheritance']:

print(f"{rel[0]} inherits {rel[1]}")

for rel in relationships['interface\_implementation']:

print(f"{rel[0]} implements {rel[1]}")

visualize\_relationships(relationships)

if \_\_name\_\_ == "\_\_main\_\_":

main()

````

---

## 2. Patch for `groovy\_analyzer.py`

Replace your entire file with the following:

````python

import os

import re

import json

import sys

from graphviz import Digraph

RELATIONSHIP\_KEYS = [

'aggregation',

'composition',

'strong\_composition',

'association',

'inheritance',

'interface\_implementation'

]

def find\_groovy\_files(root\_dir):

groovy\_files = []

for dirpath, \_, filenames in os.walk(root\_dir):

for fname in filenames:

if fname.endswith('.groovy'):

groovy\_files.append(os.path.join(dirpath, fname))

return groovy\_files

def analyze\_groovy\_file(filepath, relationships, class\_names):

with open(filepath, "r", encoding="utf-8") as f:

source = f.read()

# Remove comments

source = re.sub(r'//.\*?$|/\\*.\*?\\*/', '', source, flags=re.DOTALL | re.MULTILINE)

# Find all class/interface definitions

class\_pattern = re.compile(

r'(class|interface)\s+(\w+)(?:\s+extends\s+(\w+))?(?:\s+implements\s+([\w,\s]+))?', re.MULTILINE)

classes = class\_pattern.findall(source)

for kind, class\_name, parent, interfaces in classes:

class\_names.add(class\_name)

if parent:

relationships['inheritance'].append((class\_name, parent))

if interfaces:

for iface in [i.strip() for i in interfaces.split(',') if i.strip()]:

relationships['interface\_implementation'].append((class\_name, iface))

def analyze\_groovy\_fields\_and\_associations(filepath, relationships, class\_names):

with open(filepath, "r", encoding="utf-8") as f:

source = f.read()

# Remove comments

source = re.sub(r'//.\*?$|/\\*.\*?\\*/', '', source, flags=re.DOTALL | re.MULTILINE)

# Groovy allows def and type inference, but we'll look for explicit types and def

class\_body\_pattern = re.compile(

r'(class|interface)\s+(\w+)[^{]\*\{((?:[^{}]|\{[^{}]\*\})\*)}', re.DOTALL)

for match in class\_body\_pattern.finditer(source):

kind, class\_name, body = match.groups()

# Fields (composition/aggregation)

field\_pattern = re.compile(r'(def|\w+)\s+(\w+)\s\*(=|;)', re.MULTILINE)

for field\_type, var\_name, \_ in field\_pattern.findall(body):

if field\_type in class\_names:

if re.search(r'List<\s\*' + field\_type + r'\s\*>', body) or var\_name.endswith('s'):

relationships['aggregation'].append((class\_name, field\_type, var\_name))

else:

relationships['composition'].append((class\_name, field\_type, var\_name))

# Method parameters (association)

method\_pattern = re.compile(r'(?:def|\w+)\s+\w+\s\*\(([^)]\*)\)', re.MULTILINE)

for params in method\_pattern.findall(body):

for param in params.split(','):

param = param.strip()

if not param:

continue

parts = param.split()

if len(parts) == 2:

param\_type, param\_name = parts

if param\_type in class\_names:

relationships['association'].append((class\_name, param\_type, param\_name))

# Local variables in methods (association)

local\_var\_pattern = re.compile(r'(def|\w+)\s+(\w+)\s\*=\s\*new\s+(\w+)\s\*\(', re.MULTILINE)

for var\_type, var\_name, new\_type in local\_var\_pattern.findall(body):

if new\_type in class\_names:

relationships['association'].append((class\_name, new\_type, var\_name))

def visualize\_relationships(relationships, output\_file='groovy\_class\_diagram'):

dot = Digraph(comment='Groovy Class Diagram', format='png')

dot.attr(rankdir='LR')

# Collect all class names

class\_names = set()

for rels in relationships.values():

for rel in rels:

class\_names.add(rel[0])

class\_names.add(rel[1])

# Add nodes for each class

for cls in class\_names:

dot.node(cls, cls, shape='box')

# Add edges for each relationship type, showing variable name if present

for a, b, var in relationships.get('composition', []):

dot.edge(a, b, label=f"{var} (composes)", arrowhead='diamond', color='green')

for a, b, var in relationships.get('strong\_composition', []):

dot.edge(a, b, label=f"{var} (strong comp.)", arrowhead='diamond', color='red', penwidth='2')

for a, b, var in relationships.get('aggregation', []):

dot.edge(a, b, label=f"{var} (aggregates)", arrowhead='odiamond', color='orange')

for a, b, var in relationships.get('association', []):

dot.edge(a, b, label=f"{var} (associates)", style='dotted', color='purple')

for a, b in relationships.get('inheritance', []):

dot.edge(b, a, label='inherits', arrowhead='onormal', color='black')

for a, b in relationships.get('interface\_implementation', []):

dot.edge(b, a, label='implements', style='dashed', color='blue')

dot.render(output\_file, view=True)

print(f"Diagram saved to {output\_file}.png")

# --- Impact Analysis Helpers ---

def save\_relationships\_snapshot(relationships, filename='relationships\_snapshot.json'):

with open(filename, 'w') as f:

json.dump(relationships, f, indent=2)

print(f"Snapshot saved to {filename}")

def load\_relationships\_snapshot(filename='relationships\_snapshot.json'):

if not os.path.exists(filename):

return None

with open(filename) as f:

return json.load(f)

def compare\_relationships(old, new):

print("\n--- Impact Analysis ---")

for rel\_type in RELATIONSHIP\_KEYS:

old\_set = set(tuple(x) for x in old.get(rel\_type, []))

new\_set = set(tuple(x) for x in new.get(rel\_type, []))

removed = old\_set - new\_set

added = new\_set - old\_set

if removed:

print(f"Removed {rel\_type}: {removed}")

if added:

print(f"Added {rel\_type}: {added}")

print("--- End of Impact Analysis ---\n")

def main():

relationships = {k: [] for k in RELATIONSHIP\_KEYS}

class\_names = set()

groovy\_files = find\_groovy\_files('test\_samples')

# First pass: collect all class names and inheritance/interfaces

for filename in groovy\_files:

try:

analyze\_groovy\_file(filename, relationships, class\_names)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

# Second pass: collect fields and associations

for filename in groovy\_files:

try:

analyze\_groovy\_fields\_and\_associations(filename, relationships, class\_names)

except Exception as e:

print(f"Error analyzing {filename}: {e}")

# Remove duplicates and sort for consistent output

for k in relationships:

relationships[k] = sorted(list(set(relationships[k])))

# --- Snapshot/Compare logic ---

if len(sys.argv) > 1 and sys.argv[1] == "snapshot":

save\_relationships\_snapshot(relationships)

return

elif len(sys.argv) > 1 and sys.argv[1] == "compare":

old = load\_relationships\_snapshot()

if old is None:

print("No snapshot found. Run with 'snapshot' first.")

else:

compare\_relationships(old, relationships)

import pprint

pprint.pprint(relationships)

print("\n--- Relationships with variable names ---")

for rel in relationships['composition']:

print(f"{rel[0]} composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['strong\_composition']:

print(f"{rel[0]} strong composes {rel[1]} via variable '{rel[2]}'")

for rel in relationships['aggregation']:

print(f"{rel[0]} aggregates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['association']:

print(f"{rel[0]} associates {rel[1]} via variable '{rel[2]}'")

for rel in relationships['inheritance']:

print(f"{rel[0]} inherits {rel[1]}")

for rel in relationships['interface\_implementation']:

print(f"{rel[0]} implements {rel[1]}")

visualize\_relationships(relationships)

if \_\_name\_\_ == "\_\_main\_\_":

main()

````

---

\*\*You can now use `snapshot` and `compare` commands in both analyzers, and commented code will never be counted as a relationship.\*\*

KamleshKG: great .. want suggestion this looks as single user strategy .... if want to apply as service in corporate arena ....... wht could be best strategy where we keep audit, observability , report ...basically end to end ..where he gets impact analysis ...... think dont reply immediately

GitHub Copilot:

KamleshKG: great .. want suggestion this looks as single user strategy .... if want to apply as service in corporate arena ....... wht could be best strategy where we keep audit, observability , report ...basically end to end ..where he gets impact analysis ...... think dont reply immediately

GitHub Copilot: Certainly! Here’s a \*\*strategic outline\*\* for scaling your analyzer from a single-user tool to a robust, enterprise-ready \*\*impact analysis service\*\*:

---

## 1. \*\*Architecture: Service-Oriented & Scalable\*\*

- \*\*Central Service/API:\*\*

Deploy the analyzer as a RESTful microservice (e.g., using FastAPI, Flask, Spring Boot, etc.).

- \*\*Database Backend:\*\*

Use a relational DB (PostgreSQL, MySQL) to store:

- Snapshots of relationships (with timestamps, user, branch, commit hash, etc.)

- Audit logs of all analyses and changes

- User/project metadata

---

## 2. \*\*Integration Points\*\*

- \*\*CI/CD Integration:\*\*

Integrate with GitHub Actions, GitLab CI, Jenkins, or Azure DevOps to trigger analysis on pull requests, merges, or scheduled jobs.

- \*\*IDE Plugins:\*\*

Optional: Provide VS Code/IntelliJ plugins to show impact analysis directly in the developer’s IDE.

- \*\*Web Dashboard:\*\*

Build a dashboard (React, Angular, etc.) for:

- Visualizing class diagrams and relationship changes over time

- Browsing audit logs and reports

- Searching and filtering by user, project, or time

---

## 3. \*\*Audit & Observability\*\*

- \*\*Audit Trail:\*\*

Every analysis run logs:

- Who triggered it

- What files/branches/commits were analyzed

- What relationships changed

- Impact analysis results

- \*\*Observability:\*\*

- Use logging frameworks (ELK, Grafana, Prometheus) for metrics and error tracking.

- Expose health and metrics endpoints for monitoring.

---

## 4. \*\*Reporting & Notifications\*\*

- \*\*Automated Reports:\*\*

- Generate and email (or Slack/Teams) impact analysis reports to relevant teams on every code change.

- Allow users to subscribe to changes in specific modules/classes.

- \*\*Change Summaries:\*\*

- Summarize what relationships were added/removed/modified, and which files/classes are impacted.

---

## 5. \*\*Security & Access Control\*\*

- \*\*Authentication/Authorization:\*\*

- Integrate with corporate SSO (OAuth, LDAP, etc.).

- Control who can trigger analyses, view reports, or access audit logs.

---

## 6. \*\*Extensibility\*\*

- \*\*Plugin System:\*\*

- Allow new language analyzers or custom rules to be plugged in.

- \*\*APIs:\*\*

- Expose APIs for other tools to query relationship data or trigger analyses.

---

## 7. \*\*Example Workflow\*\*

1. \*\*Developer pushes code\*\* → CI triggers analyzer service.

2. \*\*Analyzer runs\*\*, stores relationships, compares with previous snapshot.

3. \*\*Impact report generated\*\* (with links to affected files/classes).

4. \*\*Audit log entry created\*\* (who, what, when, result).

5. \*\*Notification sent\*\* to developer/team.

6. \*\*Dashboard updated\*\* for traceability and review.

---

## 8. \*\*Tech Stack Suggestions\*\*

- \*\*Backend:\*\* Python (FastAPI), Java (Spring Boot), or Node.js

- \*\*Database:\*\* PostgreSQL or MySQL

- \*\*Frontend:\*\* React or Angular

- \*\*CI/CD:\*\* GitHub Actions, Jenkins, etc.

- \*\*Observability:\*\* ELK Stack, Prometheus, Grafana

- \*\*Messaging:\*\* Slack, Teams, Email

---

## 9. \*\*Benefits\*\*

- \*\*Traceability:\*\* Full audit trail of codebase evolution.

- \*\*Proactive Warnings:\*\* Developers see impact before merging.

- \*\*Compliance:\*\* Satisfies audit and change management requirements.

- \*\*Collaboration:\*\* Teams can review and discuss impact reports.

---

\*\*Summary:\*\*

Move from a CLI tool to a service with persistent storage, CI/CD and IDE integration, dashboards, audit logs, and notifications. This enables end-to-end impact analysis, observability, and reporting at scale for enterprise use.

If you want a sample architecture diagram or a more detailed implementation plan, let me know!

KamleshKG: give strategy left shift approach before cicd , bitbucket checkin ... we warn , inform developer of wht is happeing and give him complete code analyzer realtionship digram before and after and impact like we are doing from analyzer.py ... strategy at enterprise level ..left shift approach ...... so reviewer is confident of wht going on and new developer know wht is the codebase to study

GitHub Copilot: Absolutely! Here’s a \*\*left-shift, enterprise-level strategy\*\* for code relationship and impact analysis, integrating with developer workflows \*\*before CI/CD\*\* (e.g., at Bitbucket/GitHub check-in), to maximize code quality, transparency, and onboarding:

---

## \*\*1. Left-Shift Integration Points\*\*

- \*\*Pre-commit/Pre-push Hooks:\*\*

Integrate analyzer as a Git hook (pre-commit or pre-push) so developers get instant feedback before code leaves their machine.

- \*\*Pull Request Checks:\*\*

Run analyzer as a required check in Bitbucket/GitHub/GitLab pull requests. Block merges if critical architectural changes are detected without review.

- \*\*IDE Plugins:\*\*

Optional: Provide VS Code/IntelliJ plugin to visualize relationships and impact as developers code.

---

## \*\*2. Developer Workflow\*\*

1. \*\*Developer makes code changes.\*\*

2. \*\*On commit/push or PR creation:\*\*

- Analyzer runs automatically.

- Generates:

- \*\*Current relationship diagram\*\* (before and after change)

- \*\*Impact analysis report\*\* (added/removed/changed relationships)

- \*\*Warns developer\*\* in terminal, IDE, or PR UI about:

- What relationships are impacted

- What classes/files are affected

- Visual diff of diagrams (before/after)

- \*\*Requires developer acknowledgment\*\* or reviewer approval for significant changes.

---

## \*\*3. Reporting & Observability\*\*

- \*\*Store snapshots\*\* of relationships in a central database (with commit hash, author, timestamp).

- \*\*Audit trail\*\* for all changes and analyses.

- \*\*Web dashboard\*\* for:

- Browsing relationship diagrams over time

- Viewing impact reports for any commit/branch

- Searching for dependencies or affected modules

- \*\*Notifications\*\* (Slack, Teams, Email) for high-impact changes.

---

## \*\*4. Reviewer & Onboarding Benefits\*\*

- \*\*Reviewers\*\* see a clear, visual summary of what changed in the codebase’s architecture, not just lines of code.

- \*\*New developers\*\* can:

- Explore the codebase visually

- See how classes interact

- Understand the impact of recent changes

- Search for examples of patterns or anti-patterns

---

## \*\*5. Enterprise Strategy Summary\*\*

- \*\*Automate\*\*: Analyzer runs on every commit, push, and PR.

- \*\*Visualize\*\*: Always generate before/after diagrams and impact reports.

- \*\*Warn\*\*: Block or warn on risky architectural changes.

- \*\*Audit\*\*: Store all results for traceability and compliance.

- \*\*Educate\*\*: Use diagrams and reports for onboarding and documentation.

- \*\*Integrate\*\*: Make analyzer part of the standard dev toolchain (hooks, CI, IDE).

---

## \*\*6. Example Flow\*\*

1. \*\*Dev runs `git commit` or opens PR.\*\*

2. \*\*Analyzer runs:\*\*

- Loads previous snapshot (from DB or last commit)

- Analyzes current state

- Compares and generates:

- Relationship diagram (before/after)

- Impact report (added/removed/changed)

- \*\*Shows results in terminal/PR UI\*\*

- \*\*Blocks push/merge if needed\*\*

3. \*\*Reviewer sees diagrams and impact in PR.\*\*

4. \*\*All data stored for audit and onboarding.\*\*

---

## \*\*7. Tech Stack Suggestions\*\*

- \*\*Analyzer Service:\*\* Python/Java microservice (as you have)

- \*\*Database:\*\* PostgreSQL/MySQL for snapshots and audit

- \*\*Frontend:\*\* React/Angular dashboard

- \*\*CI/CD Integration:\*\* Bitbucket Pipelines, GitHub Actions, GitLab CI

- \*\*IDE Plugins:\*\* VS Code, IntelliJ (optional)

- \*\*Notifications:\*\* Slack, Teams, Email

---

\*\*This approach empowers developers and reviewers, increases codebase transparency, and makes onboarding and code reviews much more effective.\*\*

It’s a true left-shift, “shift-left” DevOps practice for architecture and code quality!

KamleshKG: 1. priority - to understand exisitng codebase .. new developer, experience developer , guest ....with realtionships atleast gets wht is in it

2. changes done compare with exisitng code ..... to give impact

now strategise in enterprise how two steps ... each 1 alone and each one idependent also ... as a service with all security, audit

GitHub Copilot:

KamleshKG: priority - to understand exisitng codebase .. new developer, experience developer , guest ....with realtionships atleast gets wht is in it

changes done compare with exisitng code ..... to give impact

now strategise in enterprise how two steps ... each 1 alone and each one dependent also ...both as a service with all security, audit

GitHub Copilot: Certainly! Here’s a \*\*strategy for enterprise-level code relationship analysis as a service\*\*, addressing both priorities \*\*independently and together\*\*, with security and audit in mind:

---

## \*\*Step 1: Codebase Understanding Service (Relationship Explorer)\*\*

\*\*Goal:\*\*

Enable any developer (new, experienced, guest) to quickly understand the structure and relationships in the codebase.

\*\*How to Implement:\*\*

- \*\*Service/API:\*\*

Deploy a microservice that scans the codebase and builds relationship diagrams (class, module, package, etc.).

- \*\*Web Dashboard:\*\*

Provide a UI for interactive exploration of relationships (search, filter, zoom, time-travel).

- \*\*Access Control:\*\*

Integrate with SSO (LDAP/OAuth) to restrict access by user role (guest, dev, lead).

- \*\*Audit:\*\*

Log who accessed what, when, and what was viewed/exported.

- \*\*Reporting:\*\*

Allow export of diagrams/reports for onboarding or documentation.

\*\*Independent Use:\*\*

- Can be used standalone for onboarding, architecture review, or documentation.

---

## \*\*Step 2: Change Impact Analysis Service\*\*

\*\*Goal:\*\*

On every code change, compare new relationships with previous state and report the impact.

\*\*How to Implement:\*\*

- \*\*Service/API:\*\*

Expose endpoints to:

- Upload new code snapshot (or auto-trigger on commit/PR)

- Compare with previous snapshot

- Generate impact report (added/removed/changed relationships)

- \*\*Integration:\*\*

- Integrate with SCM (Bitbucket, GitHub, GitLab) as a pre-merge or pre-push check.

- Optionally, provide CLI/IDE plugin for local use.

- \*\*Notifications:\*\*

Send impact reports to developers/reviewers via email, Slack, or PR comments.

- \*\*Audit:\*\*

Log all analyses, who triggered them, results, and actions taken.

- \*\*Security:\*\*

Only authorized users/services can trigger or view analyses.

\*\*Independent Use:\*\*

- Can be used standalone for code review, CI/CD, or compliance.

---

## \*\*Step 3: Combined Workflow (End-to-End Service)\*\*

\*\*How They Work Together:\*\*

- \*\*On codebase scan:\*\*

Relationship Explorer builds and stores the current state.

- \*\*On code change:\*\*

Impact Analysis Service compares new state to previous, generates report, and updates the explorer.

- \*\*Unified Dashboard:\*\*

Users can view both the current structure and the history of changes/impacts.

- \*\*Audit & Security:\*\*

All actions are logged; access is role-based and monitored.

---

## \*\*Enterprise Features\*\*

- \*\*Authentication/Authorization:\*\*

SSO integration, role-based access (admin, dev, guest, auditor).

- \*\*Audit Logging:\*\*

Every scan, comparison, and access is logged with user, timestamp, and action.

- \*\*Data Retention:\*\*

Store snapshots and reports for compliance and traceability.

- \*\*Observability:\*\*

Metrics, health checks, and error logging (integrate with ELK, Prometheus, etc.).

- \*\*Scalability:\*\*

Stateless services, scalable storage (cloud DB, object storage for diagrams).

- \*\*APIs:\*\*

Well-documented REST/gRPC APIs for integration with other tools.

---

## \*\*Summary Table\*\*

| Service | Purpose | Used Alone? | Used Together? |

|--------------------------|--------------------------------------|-------------|---------------|

| Relationship Explorer | Understand codebase structure | Yes | Yes |

| Impact Analysis Service | Detect/report impact of code changes | Yes | Yes |

| Unified Dashboard/API | Combine both for end-to-end workflow | - | Yes |

---

## \*\*Example User Journeys\*\*

- \*\*New Developer:\*\*

Logs in, explores diagrams, understands architecture, exports docs.

- \*\*Reviewer:\*\*

Receives impact report on PR, reviews before approving merge.

- \*\*Auditor:\*\*

Checks audit logs and historical diagrams for compliance.

class\_viz\_dashboard/

│

├── sample\_files\_relation/

│ └── sample\_project/

│ ├── analyzer.py

│ ├── relationships\_snapshot.json # (auto-generated)

│ ├── Car.py

│ ├── Engine.py

│ └── Drivable.py

│

└── (other folders/files)

\*\*This approach gives you modular, secure, auditable, and scalable services for both codebase understanding and change impact—independently or together.\*\*

KamleshKG: can you complete code and folder strcutre

GitHub Copilot: