Dependency Injection with Dagger 2

now, even Daggier.

ILYA TSYMBAL
Orange Penguin, Inc.
@ilyatsymbal
+ilyatsymbal

This presentation

Dependency Injection

- Software design pattern
- SOLID (single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion)
- Separate dependency resolution from business logic
- Code reuse (DRY)
- Evolve logic separately from dependencies
- Test business logic separately

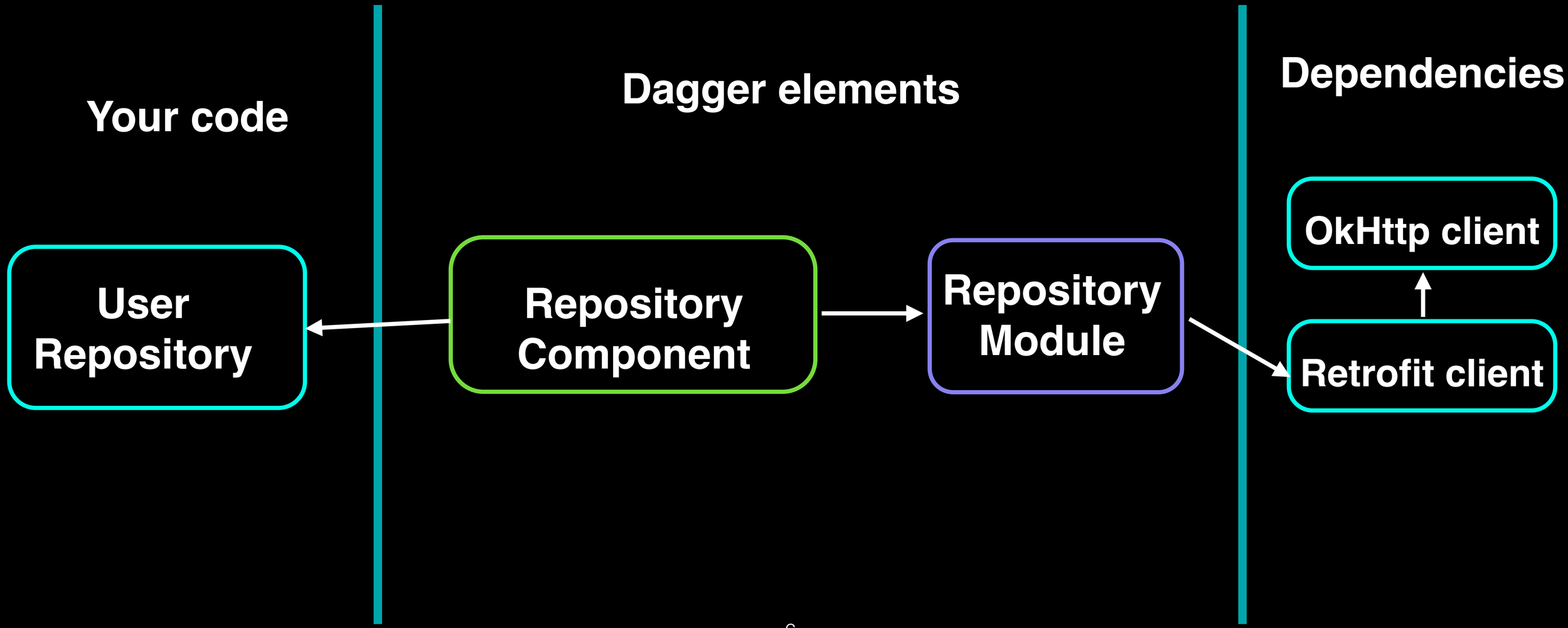
Dagger 2

- Dependency Injection library, created by Google
- Inspired by Dagger 1 library, created by Square
- Uses annotation processing with code generation
- No runtime reflection at all
- Compile time safety
- Generates readable, debuggable code

Central elements

- `@Module` – knows how to provide dependencies
- `@Component` – combines one or more modules and injects your code

Example D2 setup for Repository layer



Module

```
@Module
public class RepositoryModule {

    @Provides(name="myClient1")
    OkHttpClient provideOkHttpClient() {
        // … set up OkHttpClient
        return client;
    }

    @Provides
    public GitHubClient provideGitHubClient(OkHttpClient okHttpClient) {
      // … set up REST API client
      return gitHubClient;
    }
}
```

Dagger2 Benefit #1

Manages transitive dependencies

Component

```
@Component(modules = {RepositoryModule.class, ...})
public interface RepositoryComponent {
    void inject(UserRepository userRepository);
}
```

Your code

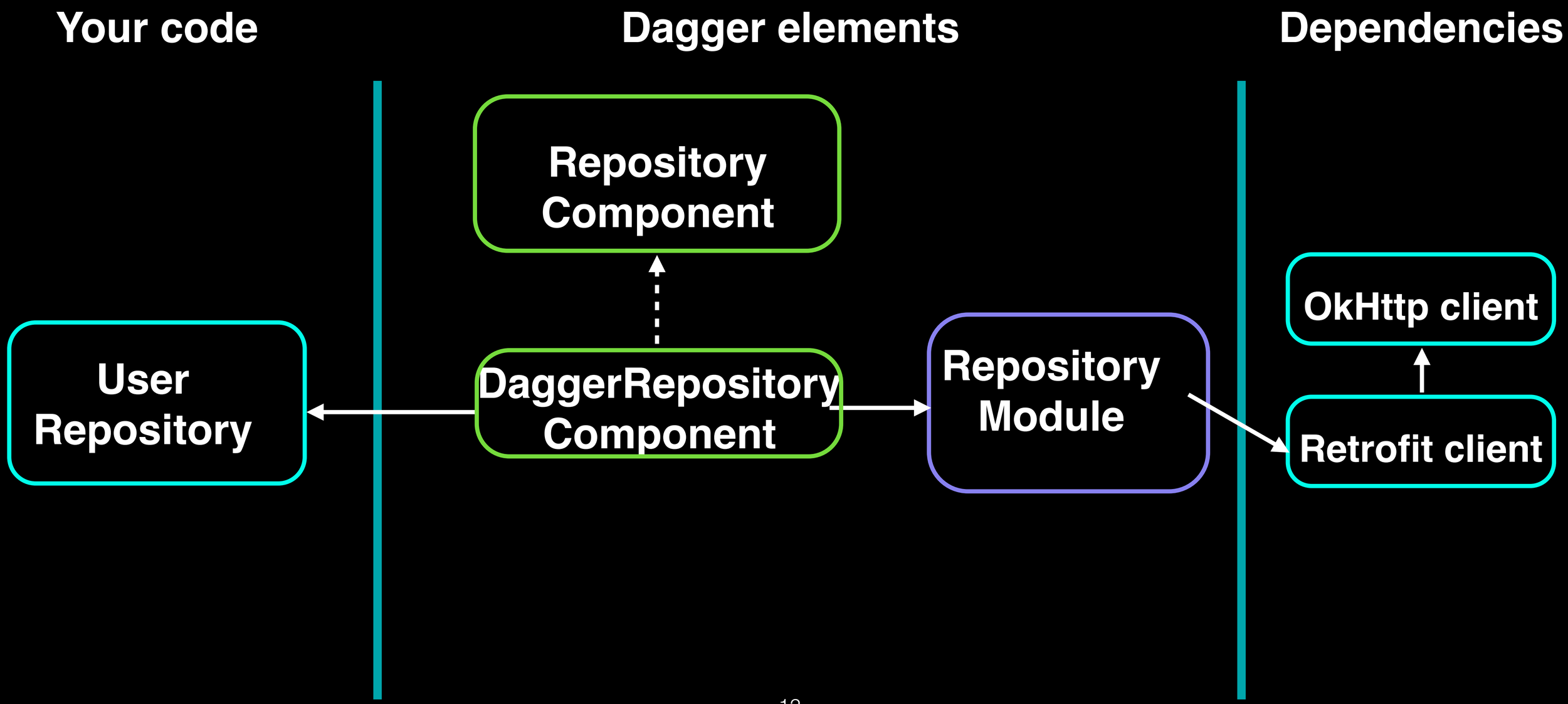
```
public class UserRepository {

    @Inject GitHubClient gitHubClient;

    public UserRepository() {
       RepositoryComponent component;
       // create or obtain component, somehow…
       component.inject(this);
    }

    public User fetchUserSyncInternal(String username)
            throws IOException {
        ApiUser apiUser = gitHubClient
                .callUser(username)
                .execute()
                .body();
        return User.fromApiUser(apiUser).build();
    }
}
```

Creating components

```
// your class that has dependencies
 public UserRepository() // constructor {
        RepositoryComponent component;
        // … create or obtain component, somehow…
        component.inject(this);
    }
```

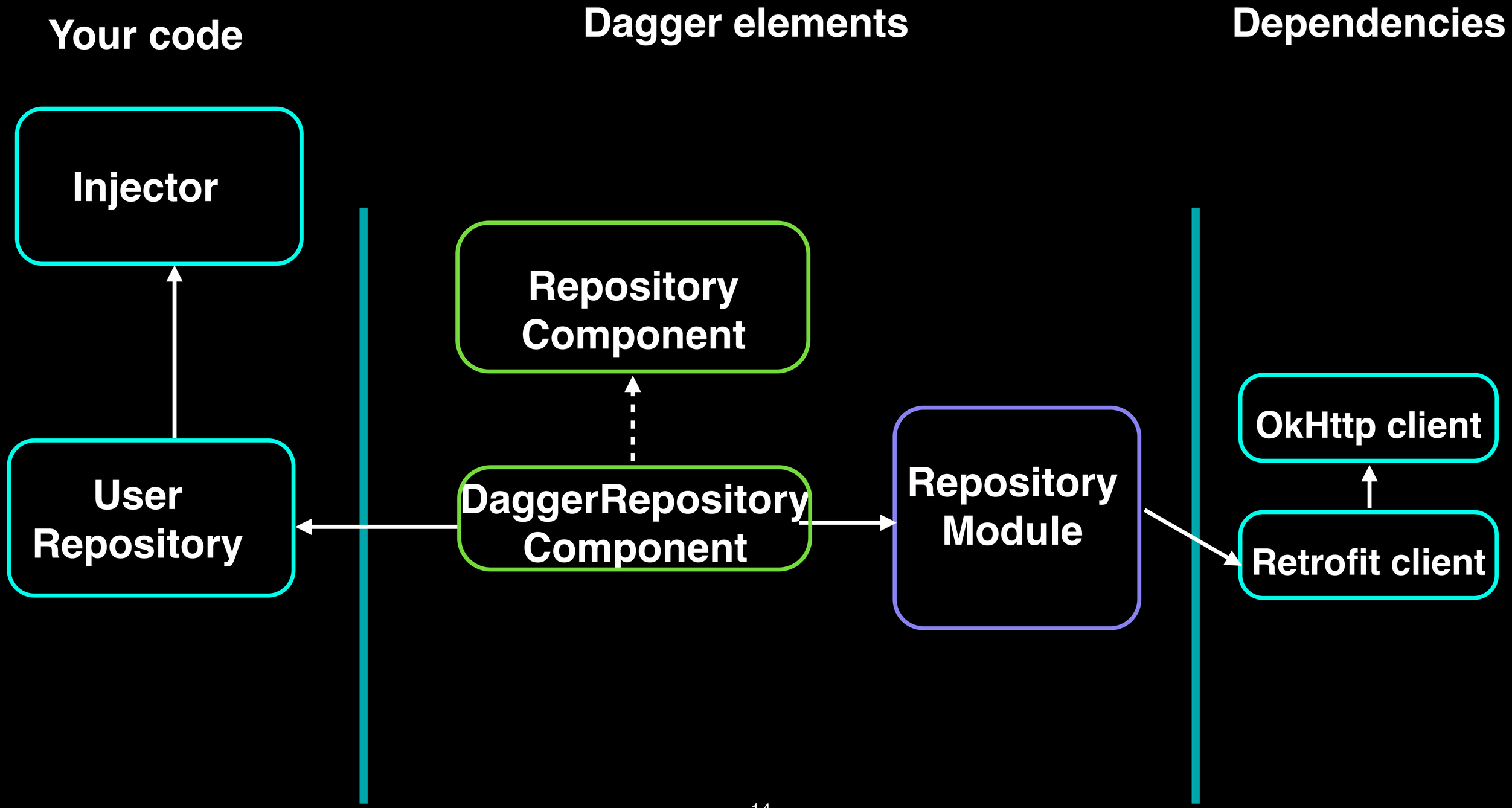
```
RepositoryComponent repositoryComponent =
        DaggerRepositoryComponent
                .builder()
                .repositoryModule(new RepositoryModule())
                .build();
```

More accurate



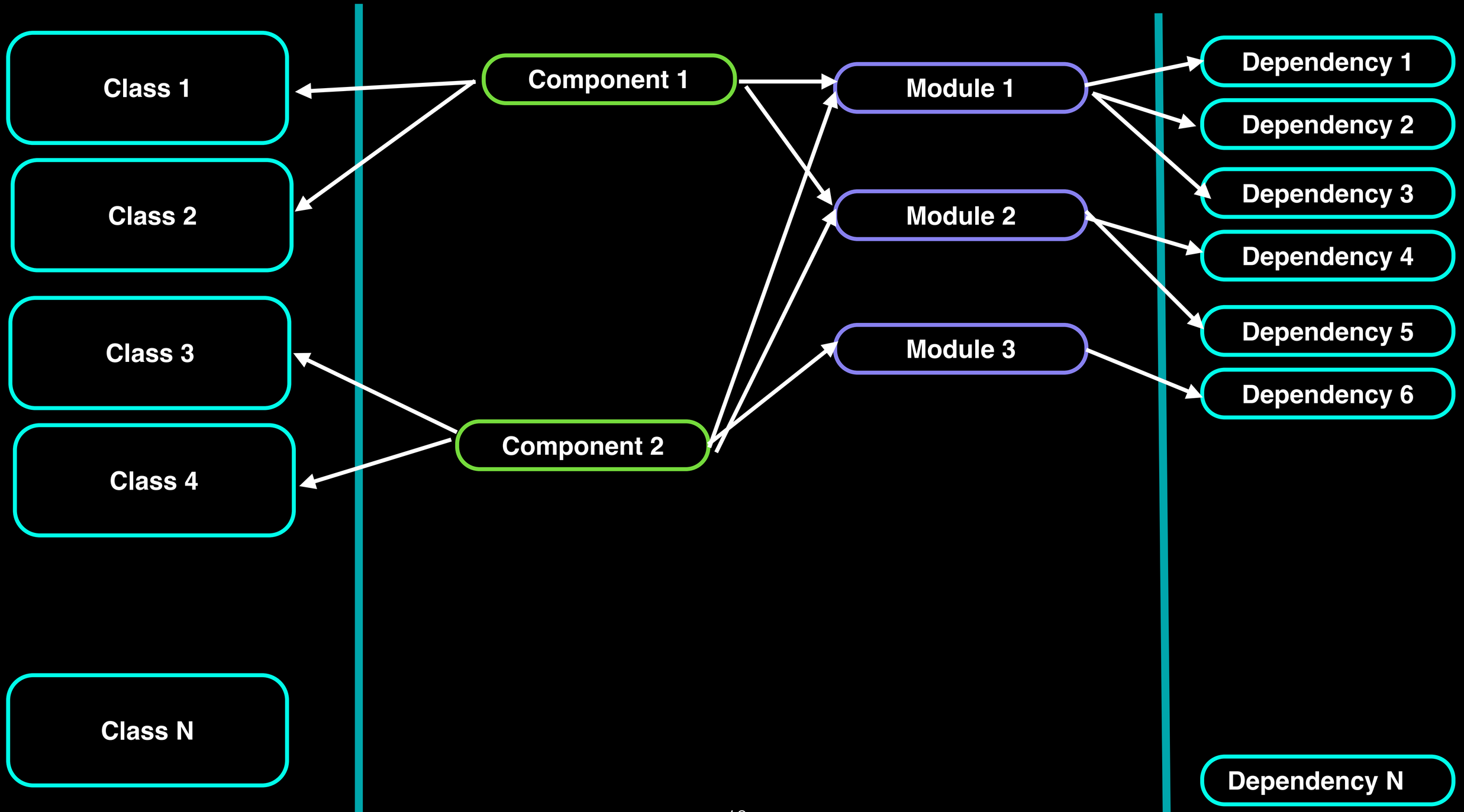
Where to put Component - Static Injector pattern

```
public class Injector {

    private static RepositoryComponent repositoryComponent;

    public static RepositoryComponent getRepositoryComponent() {
        // repository component is a singleton. Once created it is never recreated
        if (repositoryComponent == null) {
            repositoryComponent =
                    DaggerRepositoryComponent
                            .builder()
                            .repositoryModule(new RepositoryModule())
                            .build();
        }
        return repositoryComponent;
    }
}
```


Your code
Dagger elements
Dependencies
Injector
Repository Component
User Repository
DaggerRepository Component
Repository Module
OkHttp client
Retrofit client

Your code - Injection

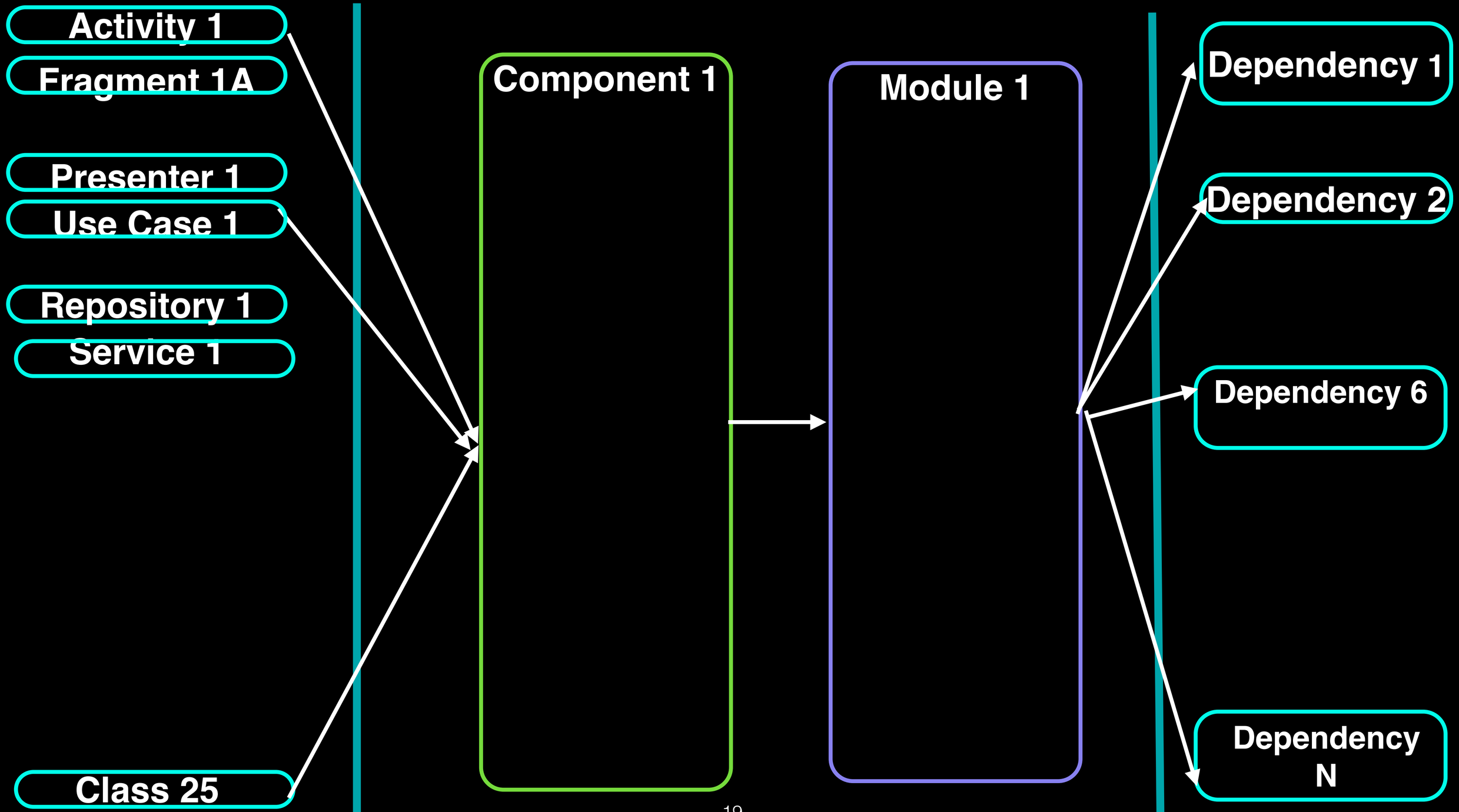
```
public class UserRepository {

    @Inject GitHubClient gitHubClient;

    public UserRepository() {
        Injector.getRepositoryComponent().inject(this);
    }

  public User fetchUserSyncInternal(String username)
            throws IOException {
        ApiUser apiUser = gitHubClient
                .callUser(username)
                .execute()
                .body();
        return User.fromApiUser(apiUser).build();
    }
}
```

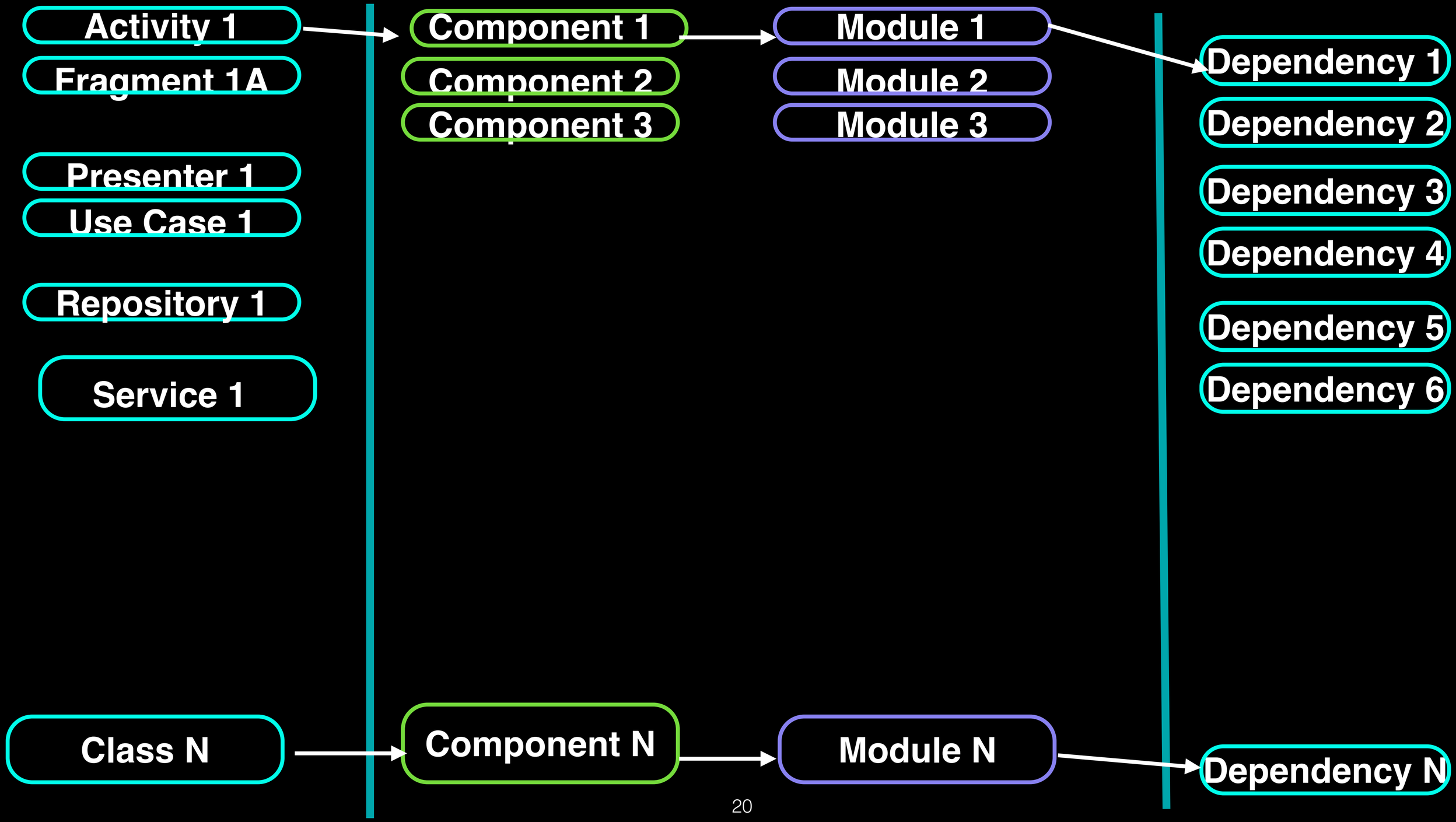

Class 1
Class 2
Class 3
Class 4
Class N
Component 1
Component 2
Module 1
Module 2
Module 3
Dependency 1
Dependency 2
Dependency 3
Dependency 4
Dependency 5
Dependency 6
Dependency N

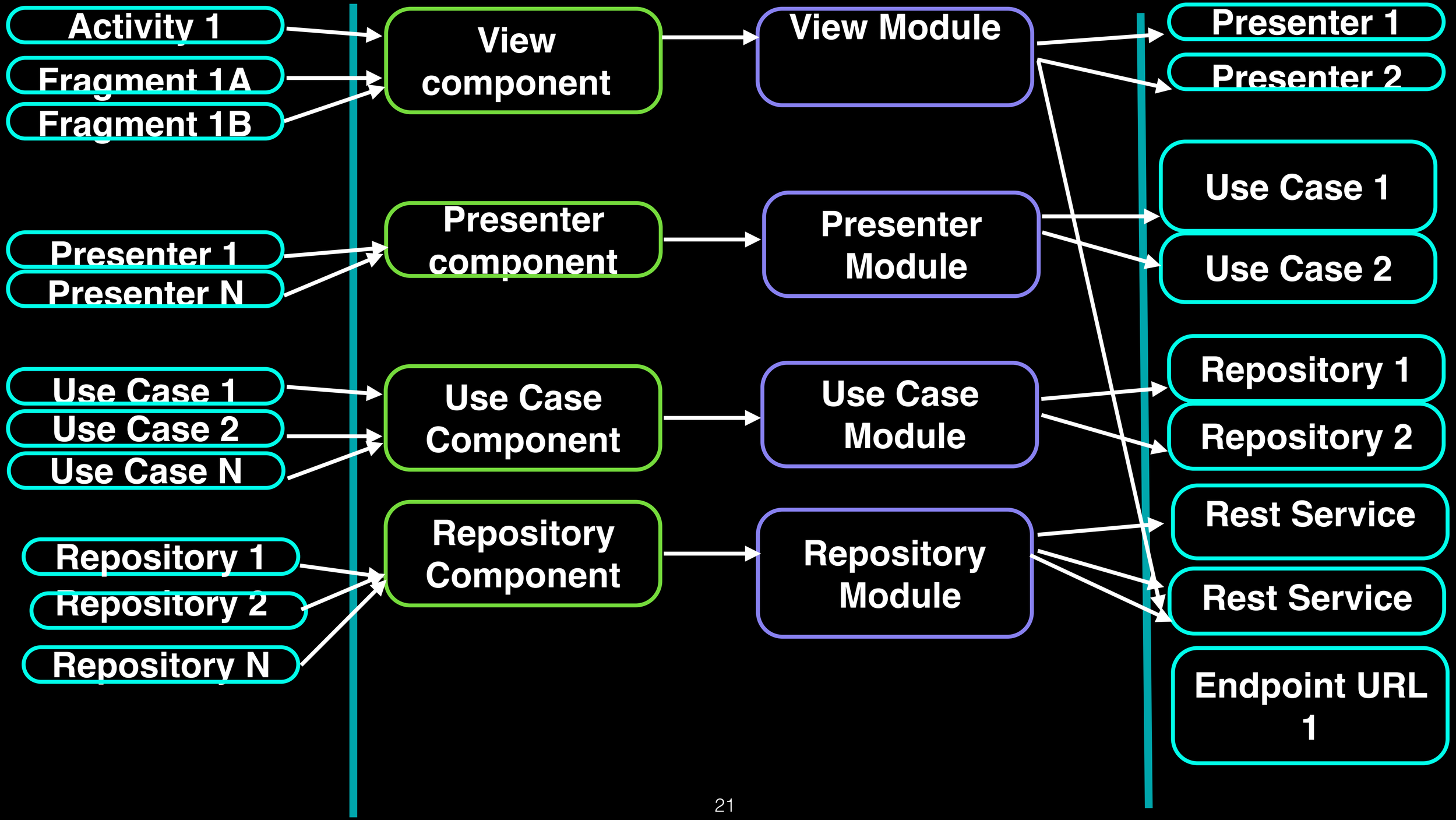
Dagger2 Benefit #2

Compose-ability of dependencies

Component and module organization


Activity 1
Fragment 1A
Presenter 1
Use Case 1
Repository 1
Service 1
Class 25
Component 1
Module 1
Dependency 1
Dependency 2
Dependency 6
Dependency N


Activity 1
Fragment 1A
Presenter 1
Use Case 1
Repository 1
Service 1
Component 1
Component 2
Component 3
Module 1
Module 2
Module 3
Dependency 1
Dependency 2
Dependency 3
Dependency 4
Dependency 5
Dependency 6
Class N
Component N
Module N
Dependency N


Activity 1
Fragment 1A
Fragment 1B
View component
View Module
Presenter 1
Presenter 2
Presenter 1
Presenter N
Presenter component
Presenter Module
Use Case 1
Use Case 2
Use Case 1
Use Case 2
Use Case N
Use Case Component
Use Case Module
Repository 1
Repository 2
Repository 1
Repository 2
Repository N
Repository Component
Repository Module
Rest Service
Rest Service
Endpoint URL 1

Testing setup

Your code - Injection

```
public class UserRepository {

    @Inject GitHubClient gitHubClient;

    public UserRepository() {
      Injector.getRepositoryComponent().inject(this);
    }

   public Call<ApiUser> callUser(String username)
           throws IOException {
       Call<ApiUser> apiUser = gitHubClient.callUser(username);
       return apiUser;
   }
}
```

Test code

```
@RunWith(MockitoJUnitRunner.class)
public class UserRepositoryTest {
    UserRepository userRepository;
    @Mock GitHubClient mockGitHubClient;

    @Before
    public void setUp() throws Exception {
        // get Injector to use mockGitHubClient, somehow...
        userRepository = new UserRepository();
    }
    @Test
    public void callUser_ShouldReturnCall()throws Exception{
          // act
          userRepository.callUser("itsymbal");
          // assert
          Mockito.verify(mockGitHubClient).callUser("itsymbal");
      }
}
```

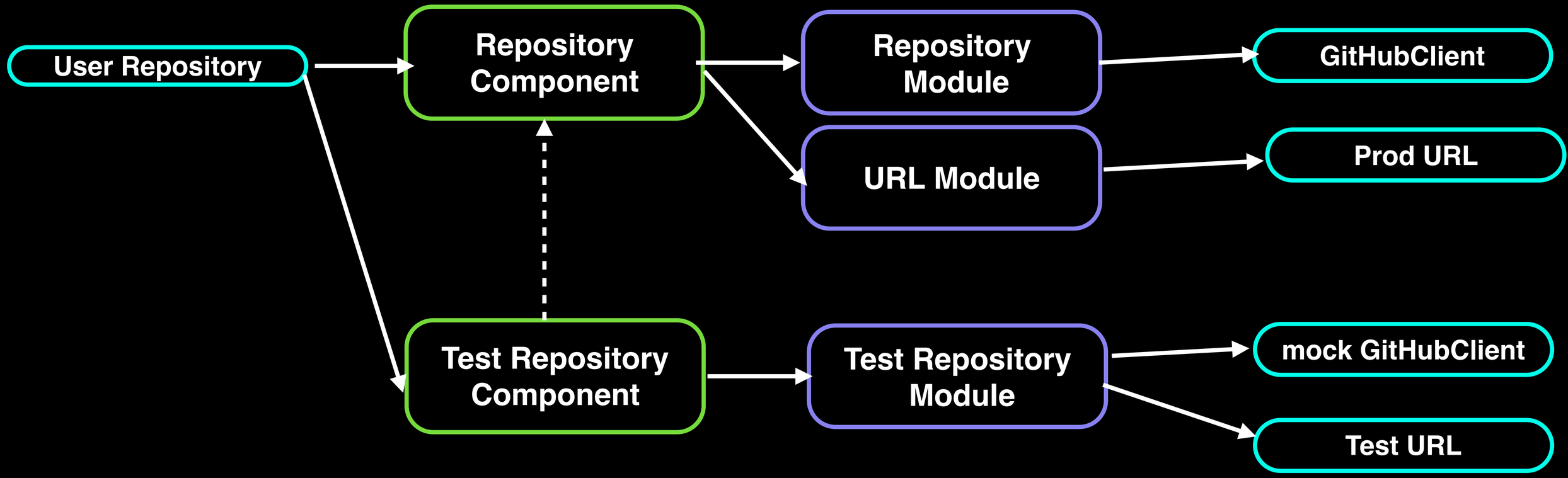
Update Injector.java

```
public class Injector {

    private static RepositoryComponent repositoryComponent;

    public static RepositoryComponent getRepositoryComponent() {
        // repository component is a singleton. Once created it is never recreated
        if (repositoryComponent == null) {
            repositoryComponent =
                    DaggerRepositoryComponent
                            .builder()
                            .repositoryModule(new RepositoryModule())
                            .build();
        }
        return repositoryComponent;
    }
}
```

Modify Injector code

```
// Injector.java
// setters allow setting a mock component from test code
public static void setRepositoryComponent(RepositoryComponent
repositoryComponent) {
    Injector.repositoryComponent = repositoryComponent;
}
```

Repository DI configuration



Production Component

```
@Component(modules = {RepositoryModule.class})
public interface RepositoryComponent {
    void inject(UserRepository userRepository);
}
```

Test Component

```
@Component(modules = {TestRepositoryModule.class})
public interface TestRepositoryComponent extends RepositoryComponent {
}
```

Production module

```
@Module
public class RepositoryModule {
    @Provides
    OkHttpClient provideOkHttpClient() {
     // set up OkHttpClient
       return client;
    }

    @Provides
    public GitHubClient provideGitHubClient(OkHttpClient okHttpClient) {

      // set up GitHubClient
    }
}
```

Test module

```
@Module
public final class TestRepositoryModule {

    GitHubClient gitHubClient = mock(GitHubClient.class);
    @Provides
    public GitHubClient provideGitHubClient() {
        return gitHubClient;
    }
}
```

ComponentUtil.java

```
public static TestRepositoryModule setUpTestRepositoryModule() {
    TestRepositoryModule testRepositoryModule = new TestRepositoryModule();

    TestRepositoryComponent testRepositoryComponent =
            DaggerTestRepositoryComponent
                    .builder()
                    .testRepositoryModule(testRepositoryModule)
                    .build();

    Injector.setRepositoryComponent(testRepositoryComponent);
    return testRepositoryModule;
}
```

Refactoring - updated Test Class

```
@Before
public void setUp() throws Exception {
   TestRepositoryModule testRepositoryModule =
       ComponentUtil.setUpTestRepositoryModule();

    mockGitHubClient = testRepositoryModule.gitHubClient;

    // configure mock object to return stub responses as needed
    // when(mockGitHubClient.callUser("itsymbal")).thenReturn(....);

    userRepository = new UserRepository();
 }
 @Test
 public void callUser_ShouldReturnCall()throws Exception{
       // act
       userRepository.callUser("itsymbal");
       // assert
       Mockito.verify(mockGitHubClient).callUser("itsymbal");
  }
```

Dagger 2 Benefit 3

Easy test setup with minimal boilerplate

References

This presentation
https://speakerdeck.com/itsymbal/dependency-injection-with-dagger-2
This presentation
https://goo.gl/tP71WP
Dagger2
https://github.com/google/dagger
Boilerplate project

This presentation