Strangling – a process example

The strangling process is pretty challenging, and if care is not taken, then you easily end in **big-ball-of-mud** where nothing works and you have lost track of the thousand places you have edited and code quality is rapidly deteriorating. *Do not go there…*

I have solved the strangling exercise, and here are some ideas to get you going. Not that this is *the solution*, more that it is inspiration to 'taking small steps' and 'keep focus' so you do not get lost big time (All tests pass all the time!)… Other 'small steps' paths are of course also possible…

The starting point and the end point

First, it is important to keep the goal in mind. Currently we have a nice monolith 'daemon' in which an implementation, PlayerServant, implements the central Player interface. It does so by fetching and storing data in a single storage via the CaveStorage interface (that is, a connector to the underlying Redis).

What we do in the strangling process is to single out one of the bounded contexts (Cave handling, message handling, player handling) and focus on the methods in PlayerServant that implements just that bounded context; and then refactor the code so these methods no longer access CaveStorage, but your service of choice.

In the example below, I have chosen *CaveService* as the selected service for introducing.

Thus the end goal is interaction that goes

> JUnit test case -> StrangledPlayerServant -> CaveService (for methods like addRoom)
>
> JUnit test case -> StrangledPlayerServant -> CaveStorage (for methods like getPlayer)

As this refactoring is basically a **ServiceTest** of PlayerServant we of course use test doubles as much as we can. In this case – all interfaces can be served by test doubles; just as the original would use 'FakeCaveStorage' as its CaveStorage implementation, we can just have a FakeCaveService implementation to avoid out-of-process calls.

Selecting a starting point.

Goal: Make a test case in which a player just get room (0,0,0) but we gradually rework the internals to fetch the room, not from CaveStorage, but from a FakeCaveService which implements a CaveServiceConnector.

I create a cave which is reading from my newly created 'strangling.cpf' file

```
@Before
public void setup() {
  String cpfFileName = Config.prependDefaultFolderForNonPathFilenames( baseFileName: "strangling.cpf");
  CaveServerFactory factory;
  PropertyReaderStrategy propertyReader;
  propertyReader = new ChainedPropertyResourceFileReaderStrategy(cpfFileName);
  factory = new StandardServerFactory(propertyReader);
  ObjectManager objMgr = new StandardObjectManager(factory);

  cave = objMgr.getCave();
  player = cave.login(TestConstants.MAGNUS_AARSKORT, TestConstants.MAGNUS_PASSWORD);
}

@Test
public void shouldGetRoom000() {
  assertThat(player.getShortRoomDescription(),
          is( value: "You are standing at the end of a road before a small brick building."));

}
```

This cpf introduce CAVE_SERVICE prefix string which is what I will use to read in the configuration of the new service.

```
# TDD of CaveService strangling

< cpf/http.cpf

CAVE_SERVICE_CONNECTOR_IMPLEMENTATION = cloud.cave.strangling.FakeCaveService
CAVE_SERVICE_SERVER_ADDRESS = localhost:9999
```

Note – the test case pass but nothing is changed.

Creating the new Player implementation: StrangledPlayerServant

I must pursuade the factory to create a new Player instance which returns my new PlayerServant implementation which interacts with the CaveService, not the CaveStorage. I do this by "inline" modifying the factory

```
// Now pursuade the Factory to create my new "strangled" implementation of PlayerServant
factory = new StandardServerFactory(propertyReader) {
  @Override
  public Player createPlayerServant(LoginResult theResult, String playerID, ObjectManager objectManager) {
    testLogger.info("method=createPlayerServant. implementationClass=StrangledPlayerServant");
    return new StrangledPlayerServant(theResult, playerID, objectManager);
  }
};
```

IntelliJ will of course complain that no such class exist. Just create it and then **boldly copy all the source code of the original PlayerServant into this class.** (Fix the name of the logger instance and a few comments here and there). Of course keeping two copies of the same source code is normally the start of hell, but in our case it is a stepping stone – keeping the old code around until we are sure the new one works.

As often in test-driven-development this class serves as scaffolding for the code to come: Method by method it will be strangled to become the new microservice oriented implementation, but still relying on the superclass to cover functionality for the 'non-strangled' part, like messages and player handling.

My 'getRoomAt000' test pass, quite obviously, nonthing has changed; but I do view the log message, verifying that indeed I am having the right code executing:

```
2021-10-22T11:29:25.223+02:00 [INFO] cloud.cave.config.StandardServerFactory :: method=createServerRequestHandler, impleme
2021-10-22T11:29:25.225+02:00 [INFO] STRANGLING :: method=createPlayerServant. implementationClass=StrangledPlayerServant
```

Preparing to fetch initial Rooms from CaveService

The constructor of PlayerServant 'refreshes from storage' which is what we need to change into refreshing partially from the CaveService. Thus on top of having a 'storage' instance variable, I also need a 'caveService' instance variable.

Getting proper access to the CaveService implementation

Now it is time to actually do something – we get access to our new CaveService in the PlayerServant. The original code in this guide was wrong – I used the factory, but you should call directly to the ObjectManager, as I do below. The trick is that the objectManager and CPF system already allows us to do so:

```
private final CaveServiceConnector caveService;
public StrangledPlayerServant(LoginResult theResult, String playerID, ObjectManager objectManager) {
  // Instance variables duplicated in superclass, but will disappear once strangling is complete
  this.ID = playerID; this.objectManager = objectManager;
  this.storage = objectManager.getCaveStorage();

  // Now, get access to the connector to the CaveService
  this.caveService = objectManager.getServiceConnector(CaveServiceConnector.class,
          StranglingConstants.CAVE_SERVICE);
  logger.info("method=constructor, action=created-caveService, caveService={}", caveService);

  // This constructor assumes the user has already been logged in.
  this.authenticationStatus = LoginResult.LOGIN_SUCCESS;
  refreshFromServices();
}
```

Making this change require a few steps – the CAVE_SERVICE constant must match the prefix in the CPF file, a CaveServiceConnector interface which implements the ExternalService interface must be in place:

```
public interface CaveServiceConnector extends ExternalService {
}
```

(The ExternalService is required for the Factory system in SkyCave), and of course our first FakeObject implementation of CaveServiceConnector:

```
public class FakeCaveService implements CaveServiceConnector {
  @Override
  public void initialize(ObjectManager objectManager, ServerConfiguration config) {
  }
  @Override
  public void disconnect() {
  }
  @Override
  public ServerConfiguration getConfiguration() {
    return null;
  }
}
```

Again, this is more or less just building scaffolding code, but we can see that the FakeCaveService is actually created from the log file when running the test case.

```
INFO] cloud.cave.strangling.StrangledPlayerServant :: method=constructor, action=created-caveService, caveService=cloud.cave.strangling.FakeCaveService@54
```

Fetching initial rooms from CaveService

Finally, it is time to rewire the StrangledPlayerServant – to request the room from the CaveService, not from the CaveStorage. I change the 'refresh' code to

```
private void refreshFromServices() {
  PlayerRecord pr = storage.getPlayerByID(ID);
  name = pr.getPlayerName();
  groupName = pr.getGroupName();
  position = pr.getPositionAsString();
  region = pr.getRegion();
  accessToken = pr.getAccessToken();

  currentRoom = caveService.getRoom(position);
}
```

And intelliJ yells as me as the method is not defined. I just use the 'Alt-Enter' in IntelliJ to let it create the method in both interface and Fake object implementation. Also note that I still use the ordinary 'storage' for player ID etc; I only strangle one service at a time!

In the Fake implementation I make something that *will break the test* just to verify that it is the proper code that is called:

```
@Override
public RoomRecord getRoom(String position) {
  RoomRecord room = new RoomRecord( description: "A Wrong Description",  creatorId: "0");
  return room;
}
```

The code fails – to prove that the room was retrieved from the Cave service, not from the original cave storage.

```
java.lang.AssertionError:
Expected: is "You are standing at the end of a road before a small brick building."
     but: was "A Wrong Description"
Expected :You are standing at the end of a road before a small brick building.
Actual   :A Wrong Description
```

Conclusion at this point

The main point is that we have now established the "chain of objects" that form the basis for the real strangling process: **Cave interacts with StrangledPlayerServant interacts with FakeCaveService (and the old CaveStorage for the other services).**

The new 'StrangledPlayerServant' can now gradually be implemented by adding more and more methods that contact the CaveService instead of the CaveStorage; and gradually just copying code (that is relevant for Rooms and only that) from my FakeCaveStorage into the FakeCaveService in a lock step manner.

Once all room related methods are working; you can then create **connector tests** for the CaveServiceConnector, that is, out-of-process tests that use TestContainer to create a real REST based CaveService, and then develop the code of the 'RealCaveService implements CaveServiceConnector' – that uses HTTP calls to query and update the underlying cave service.

A few observations

I quickly found that making the fake service use other room descriptions than the original ones made it much more obvious that my strangled player servant was fetching from the wrong service, so I ended up with

```
// Initialize the default room layout to be DIFFERENT from the FakeCaveStorage
RoomRecord entryRoom = new RoomRecord(
        description: "Strangled outside brick building.", CaveStorage.WILL_CROWTHER_ID);
this.addRoom(new Point3( x: 0,  y: 0,  z: 0).getPositionString(), entryRoom);
this.addRoom(new Point3( x: 0,  y: 1,  z: 0).getPositionString(), new RoomRecord(
        description: "Strangled deep valley.", CaveStorage.WILL_CROWTHER_ID));
this.addRoom(new Point3( x: 1,  y: 0,  z: 0).getPositionString(), new RoomRecord(
        description: "Strangled well house.", CaveStorage.WILL_CROWTHER_ID));
this.addRoom(new Point3( x: -1,  y: 0,  z: 0).getPositionString(), new RoomRecord(
        description: "Strangled hill.", CaveStorage.WILL_CROWTHER_ID));
this.addRoom(new Point3( x: 0,  y: 0,  z: 1).getPositionString(), new RoomRecord(
        description: "Strangled tall tree.", CaveStorage.WILL_CROWTHER_ID));
```

Which on second thoughts are a bit macabre.

Also I later realized that I could circumvent the CPF system by reusing the TestDoubleed factory ala:

```
@Before
public void setup() {
  // Create the factory with all doubles, but the player is the strangled variant
  // And the service connector for Cave is the FakeCaveService
  CaveServerFactory factory = new AllTestDoubleFactory() {
    @Override
    public Player createPlayerServant(LoginResult theResult, String playerID, ObjectManager objectManager) {
      return new StrangledPlayerServant(theResult, playerID, objectManager);
    }
    @Override
    public ExternalService createServiceConnector(Type interfaceType, String propertyKeyPrefix, ObjectManager objectManager) {
      // HARD CODED
      ExternalService service = new FakeCaveService();
      service.initialize( objectManager: null, new ServerConfiguration( ip: "localhost",  port: 9999));
      return service;
    }
  };
  ObjectManager objMgr = new StandardObjectManager(factory);

  cave = objMgr.getCave();
  player = cave.login(TestConstants.MAGNUS_AARSKORT, TestConstants.MAGNUS_PASSWORD);
}
```