Decoupling

Decoupling is a technique that can be used to transform code that is considered to be a potential service candidate, but it mixes layers which render making services extraction complex. Decoupling helps us separating these layers and thus reducing the service extraction complexity.

Covered topics in this document:

- Definition of services.
- Issues affecting service extractor.
- Fixing techniques.

Understanding Services

To understand what the target of the Service Map is, let's define what is a service for us. To do this, we will borrow concepts from the hexagonal architecture.

The hexagonal architecture tries to take the core application and exposes it through ports and adaptors. In this point we don't need to worry whether a core code can be represented as domain + application layer or if its code is spaghetti or not, the target is to build a mechanism that "extracts" the logic that some developer is interesting in. That is, it doesn’t matter if there are technical debts, anemic models or spaghetti design in the code, we are extracting logic that can be "relevant" (valuable) for the business and can be used as stateless services.

To fix design issues there are techniques available, like refactoring, that can help reducing the tech-debt and weak-domain design.

The next naive definitions help to understand the context of services that we want delivery.

Definition 1.1. Port is an data entry point access, as an TextBox, Http Method, File, etc. an port is related with an Boundary Class.

Definition 1.2. WebPort is a data entry point that can receive distributed objects in many formats as JSON, XML, etc.

Definition 1.3. Service is a collection of WebPorts, exposing it as distributed methods.

Definition 1.4. Microservice is a collection of the services, putting together in some concept as Invoice or Account, that has organizational meaning.

Definition 1.5. Service Oriented Architecture is a large collection of the services that are exposing as an API, could be called as Monolith API, because there is not conceptual physical separation

With these definitions we are not necessarily classifying services into stateless or stateful ones; our goal is to extract Stateless Services. This means to be able to expose some business logic through a Web Port.

Issues affecting service extractor

The Structure: How well is the code design (Deployment View).

The Code: Are there anti-patterns in the code (Logical View).

The Structure: how well is the code design deployment view

The structural analysis is a high level code diagnostic to allow classify the level of reusability of the application.

There are some architectures that are going to require an significant amount of effort to be exposed as Services.

Now let of define some concepts that will help us classify some scenarios.

Definitions

Definition 2.1. Artifact is an piece of software that you can consume as an potential-assembled component.

Definition 2.2. Well-defined artifact is an potential-assembled component that represent business logic with good level of decoupling.

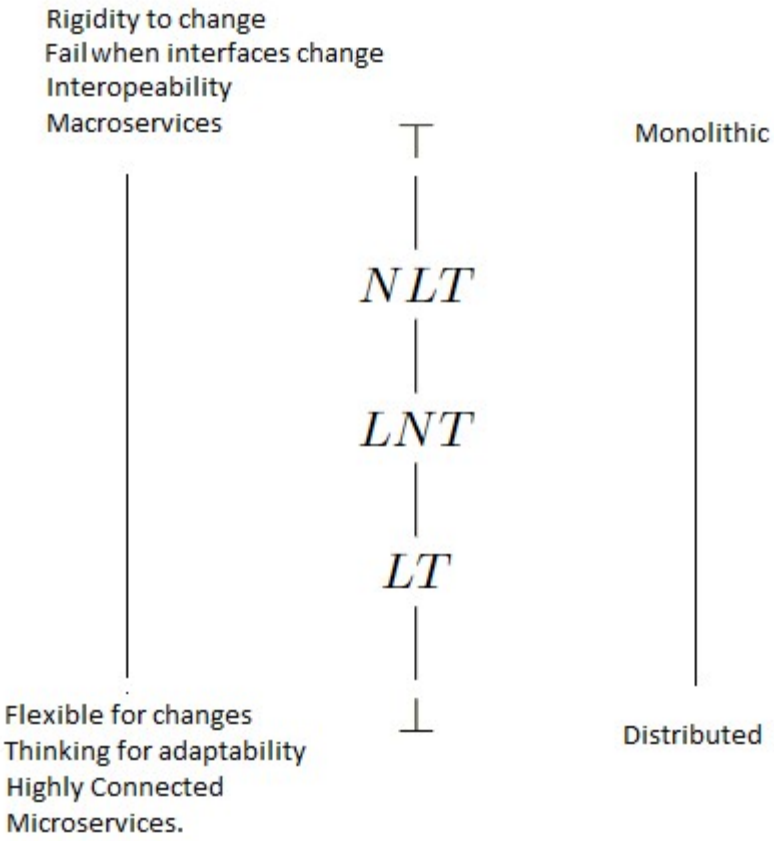
Definition 2.3. Tier is a physical separation where some artifacts can be deployed in many environments or nodes(machines).

Definition 2.4. Layer is a logic software separation where some well-defined artifacts can be used as libraries.

These definitions help us understand possible scenarios where services could be extracted using **Service Map** and the associated complexity.

Scenarios	Definition	Services Automation	Implications	Service Extractor Techniques
Not Layer Not Tier (NLT)	an NLT is an monolithic application that are put together in one single artifact.	candidates for services is low.	the extraction of the services can be hard to do.	UI Macro Services or Refinement + Code Service Extractor
Layer Not Tier (LNT)	An LNT is an monolithic application that the code is separate but it is not deployed as Client-Server.	candidates for services are medium-high.	if the code is well-defined layered the extraction is easy; otherwise it introduces a lot of complications.	Refinement + Code Services Extractor
Layer and Tier (LT)	An LT is an architecture that is physical and logical separate.	candidates to services is high.	the problems of many LT Architecture is that the don't have a separation concept.	Refinement + Micro Services extractor

This classification offers an tentative order of extraction:



The Code: Are there anti-patterns in the code logical view

Another factor that can affect the "extraction" of the code, is how the code is designed.

Corruption Layer

The code has a layer and there are some bottom layer that access directly to top layer.

Cyclic Dependency

Some class that depends of another class and generate a cycle of dependencies.

Many Boundaries one layer

When some code has not any wrapper on the boundaries or there are not use of UI Patterns, introducing high coupling in the business logic.

These anti-patterns introduce a lot of the complexity in the migrations due to it should be reduced before to migrate.

Resume chart

Anti Pattern	Consequences
Corruption Layer	Problems where front end logic exists in many back end logic,ex: an MessageBox in the data access layer
Cyclic Dependency	When you need to separate logic that depend of cyclic invocations
Two Boundaries One Layer	This anti pattern appears where you have two or more boundaries as Top or Bottom in the same code

Problem Solutions

1. Fix yourself.
2. Use an refactoring tool.

Solution Implications

Fix yourself	Refinement Tool
the migrations will be more expensive	we need to invest money on it
The decoupling can be hard depending of the anti-patterns	several cases cannot be resolved
Depends of the business logic and require knowledge	Should be a friendly tool that allow to define simples rules
Is hand made	Can be semi-automatic

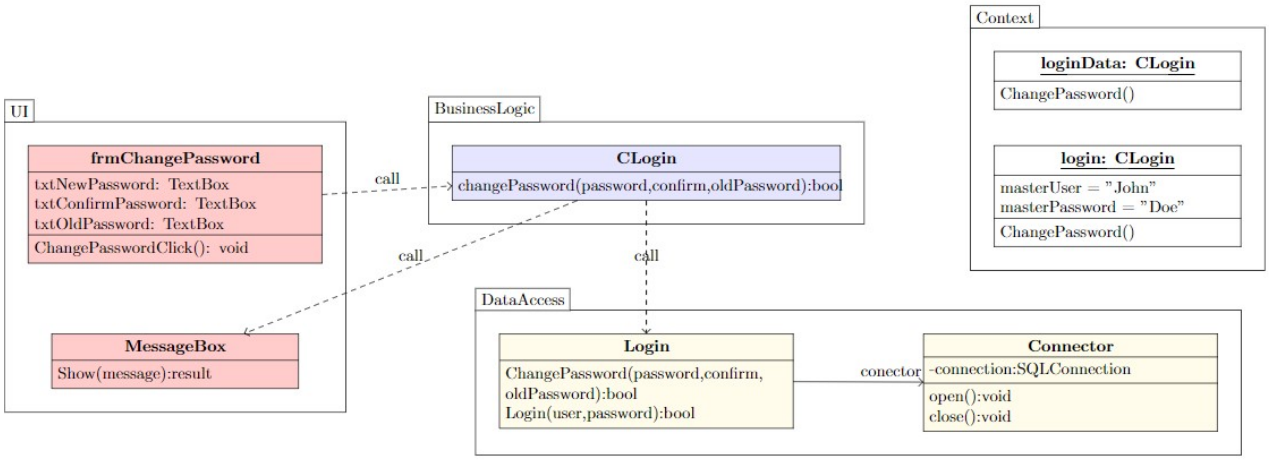
Refinement Tools (fixing techniques)

An refinement tools allow to made refactoring in the code to allow separate logic, rewrite code, with the intention of tier-ize and layer-ize the software.

Corruption Layer Proposal

Looking in the Problem

The next example is to review how we can transform the code when there are some UI Elements on the bottom layers.



Look class CLogin that belongs to Business logic but it is calling UI Layers and Data Access Layers. This kind of the design introduce the problem of the layer corruption

```
public class frmChangePassword: Form {
        public void ChangePasswordClick() {
                if(Context.Login.ChangePassword(txtNewPassword.text, txtConfirmPassword.text, oldPassword.text))
                {
                        MessageBox.Show("Change Password was changed!!!!");
                }
        }
}
```

Some particular problems that we can identify, is the use of the Context Variables, most of the time static implementations.

```
public class CLogin {

        private string masterUser;

        private string masterPassword;

        public bool ChangePassword(string password,string confirm,string oldPassword){
                var success = false;

                if(masterPassword != password){
                        MessageBox.Show("the password doesn't match");
                }
                else {
                        if(password != confirm){
                                MessageBox.Show("there are differences between password")
                        }
                        else {
                                success = context.LoginData.ChangePassword(masterUser, password);
                        }
                }
                return success;
        }
}
```

in these layer you can see that CLogin is calling UI components and also it is calling Data Access components,

```
public class Login {

        public bool ChangePassword(string user, string password) {

                var connection = Connector.Open();
                var procedure = connection.Procedure("sp_changepassword");
                procedure.Add("user",user)
                procedure.Add("password",password);
                var result = connection.execute();
                Connector.Close(connection);
                return result;
        }

}
```

Refinement Algorithms

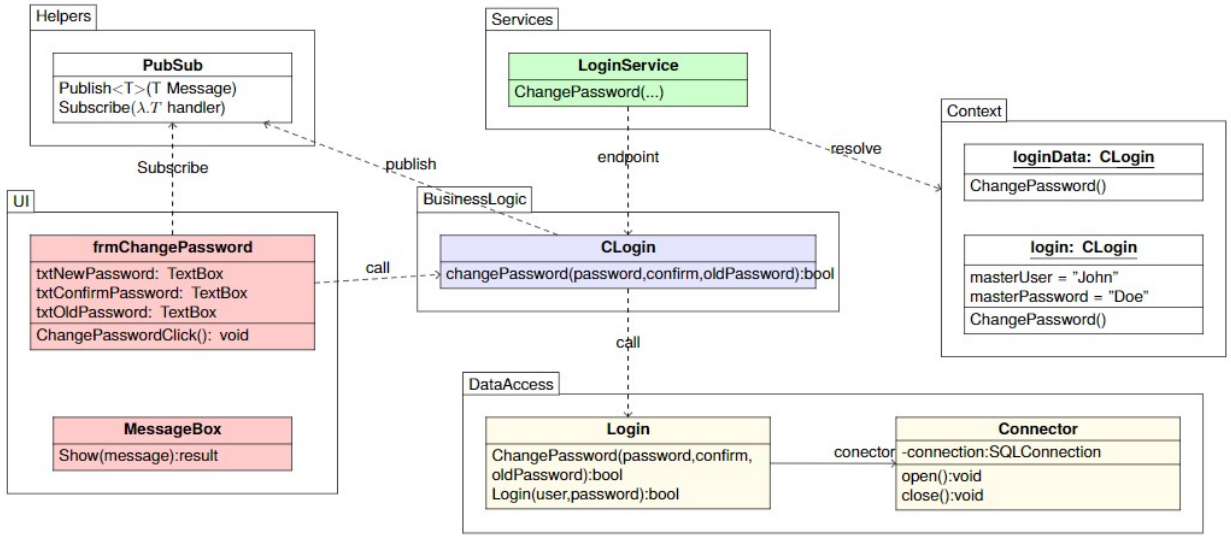
So one technique that we can use here is try to introduce some patterns that allow to decouple the code

We want to introduce some patterns as:

Pattern	Used by
Publish Subscriber	To decouple the MessageBox communication, or another notification logic

Pattern	Used by
Dependency Injection	To inject the creation of the objects without an explicit "new" allow to depend of interfaces
Weaving	To inject code allowing simplify the code statement, for example change a static declaration

Look that if we didn't decouple these part, we can not reutilize the validation business logic.



Now you can appreciate how decoupling was made, deleting front end objects from CLogin class and replacing it for a notification pattern.

```
public class CLogin {

        private string masterUser;

        private string masterPassword;

        [injected] //using dependency injection
        private LoginData loginData;

        [injected]
        private IPubSub notify; // using dependency injection

        public bool ChangePassword(string password,string confirm,string oldPassword){
                var success = false;

                if(masterPassword != password){
                        /// using publish subscriber
                        notify.publish("the password doesn't match");
                }
                else {
                        if(password != confirm){
                                notify.publish("there are differences between password")
                        }
                        else {
                                success = loginData.ChangePassword(masterUser, password);
                        }
                }
                return success;
        }
}
```

Required techniques to separate code

Technique	Details
Context Analysis	With these analysis we can obtain the scope dependencies of the methods, and also determinate what are the variables than should be provide to resolve the context of the one method, allowing transform some state-full method to a stateless method.
Refinement Tools	Refactoring tools that can detect some patterns to allow separate code, edit code or create new structures
Pattern Replacer	it is part of the refinement tool, that evaluate the code to replace some structures for the most acceptable patter
Weaving	Help to inject code to resolve dependencies and replace context variables.

Cyclic Dependencies Proposal

Two Boundaries - One Layer Refinement proposal

Looking in the problem

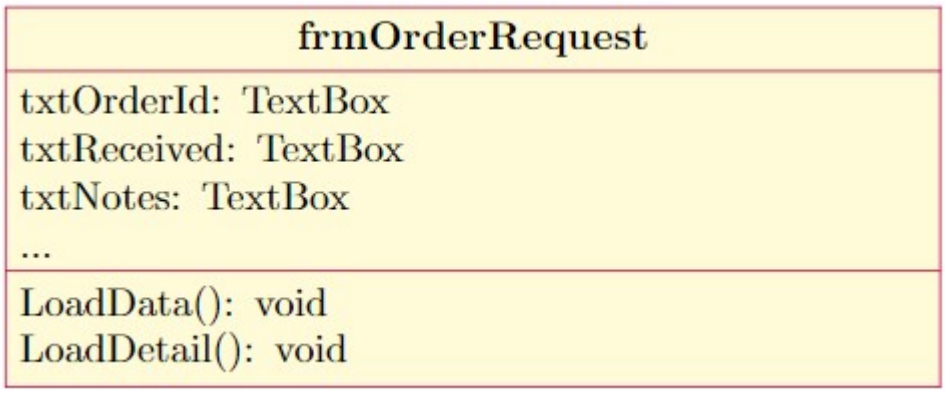
The problems to separate two kind of layers that can be exists in one code, is not simple, we need to identify the UI statement to separate the code.

The criteria to determinate if some expression can be define in another kind of the object, it is only if it is know as an output.

Well the UI can be as a port, in facts, it is human asynchrony port, so the idea is to represent what the UI objects and fields are defined in the code, and move it for another structure that contains in essential the same data.

> [!NOTE] we only want to extract the data, because the algorithms in the controls are not part of the business logic.

Looking the diagram



simplifying the logic we can see the structural form of the intention of the LoadData and LoadDetails algorithm

Algorithm 1: LoadData

Data: $T \in UI$
Result: Loading UI fields with Data Base Information
initialization;
begin
  $result \longleftarrow execute(query)$
  $T[id] \longleftarrow Transform(result[id])$
  $call \rightarrow LoadDetails$
end

Algorithm 2: LoadDetail

Data: $G \in UI.Grid$
Result: Loading Grid field with Data Base Information
begin
  $result \longleftarrow execute(query)$
  forall *item of result* **do**
    $G[id] \longleftarrow item[id]$; `// loading the grid control`
  end
end

As we can see the flow of the state could be suggestion something

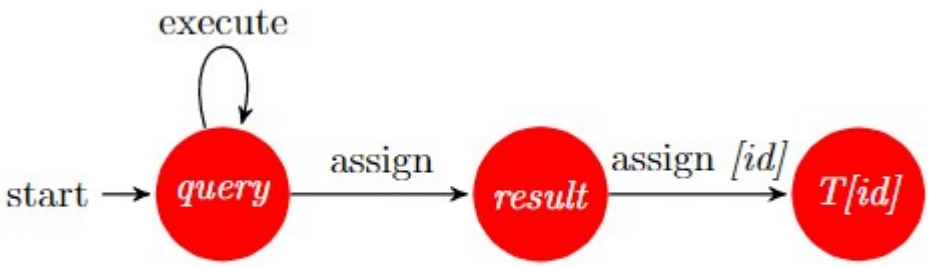


Figure 1: Flow state sequential

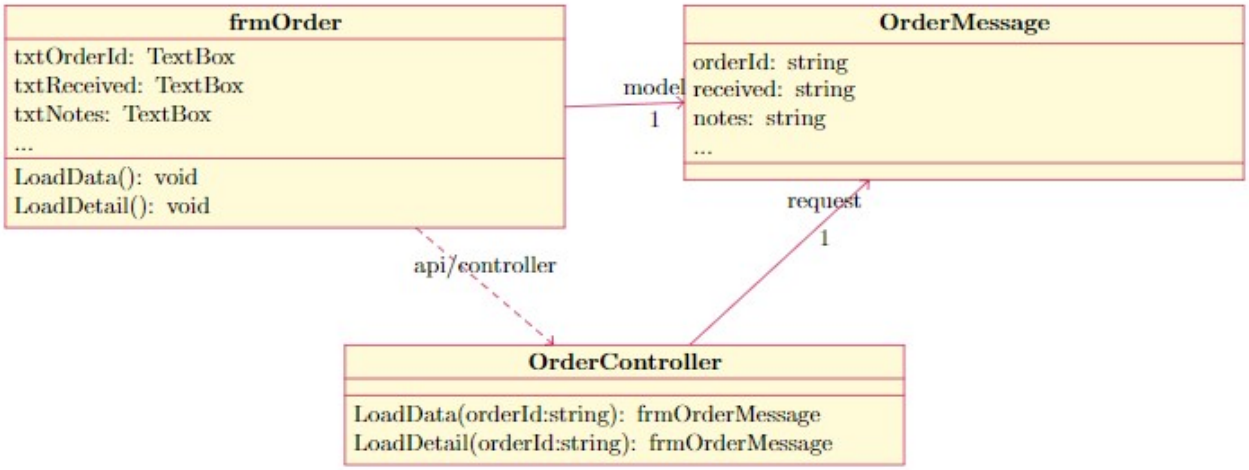
the flow is saying that T[id] is an output, so maybe we can introduce some pattern to separate the logic introducing some transfer object.

Refinement Algorithms

One technique to explore is try to separate the View Code to a simple pattern as Model View Controller.

In the controller we want to expose only methods that are stateless, and also that only do one thing, as you can see Load Data and Load Details are populating data without have any synergy between them, that means the two methods are completely autonomous, so we can expose both of them as WebPorts

Design Separation Logic



For architectures that are not well separate we propose to separate the UI Logic, Data Object (model) and the Business logic (Controller) allowing decouple the UI code.

Algorithm 3: LoadData: *Web Port*

Input: $orderId : int$
Output: $model : OrderMessage$
Result: Loading model with Data Base Information
begin
  $result \longleftarrow execute(query)$
  $model[id] \longleftarrow Transform(result[id])$
  return *model*
end

In an web port object the method as Load Data should load the information on Message that can be render as JSON or XML, in these case the method return an object as OrderMessage,

Look that these method is not calling the LoadDetails logic, because the algorithms are autonomous, that means that one method don't depend for the other can be executed as parallel.

Algorithm 4: LoadDetail: *Web Port*

Input: $orderId : int$
Output: $model : OrderDetailMessage$
Result: Loading model with Data Base Information
begin
  $result \longleftarrow execute(query)$
  forall *item of result* **do**
    $model[id] \longleftarrow Transform(item)$
  end
  return *model*
end

In these case the method is returning OrderDetailMessage

Algorithm 5: LoadData: *UI Port*

Data: $T \in UI$
Result: Loading UI fields with Data Base Information
initialization;
begin
  $result \longleftarrow OrderController.LoadData(txtOrderId)$
  $T[id] \longleftarrow model[id]$
  $call \rightarrow LoadDetails$
end

This is the part of the original method, now the method be refinement as a method that call the controller get the data and render it to the UI Port.

Algorithm 6: LoadDetail: *UI Port*

Data: $G \in UI.Grid$
Result: Loading Grid field with Data Base Information
begin
  $result \longleftarrow OrderController.LoadData(txtOrderId)$
  $G.data \longleftarrow result$; `// loading the grid control`
end

The same for the detail.

Separation Analysis

As you can see we separate the code from two kind of the ports

UI Ports: will contain statements that references to UI statements.

Web Ports: Will contain statements that references to Business Logic or Backend boundary logic.

Required techniques to separate code

Technique	Details
Context Analysis	With these analysis we can obtain the scope dependencies of the methods, and also determinate what are the variables than should be provide to resolve the context of the one method, allowing transform some state-full method to a stateless method.
Synergies Classification	With these technique we can define if the methods has some dependency in terms of the call and if they can be executed in parallel
UI Semantic	We need to detect some part of the Syntax Tree as UI Expression or Backend Expression and it can be separable
Boundary Control Flow	With these tools we can identify what are the control logic of some logic as UI or some Logic as Backend.
Refinement Tools	Refactoring tools that can detect some patterns to allow separate code, edit code or create new structures
Layering Tool	With these tools we can separate the code depending some criteria that you define, depends of your Top-Bottom layering order.