Implementing Simple Business Logic

As we saw earlier, not all business subdomains are created equal. Different subdomains

have different levels of strategic importance and complexity.

We will start with two patterns suited for rather simple business logic: transaction script and active record.

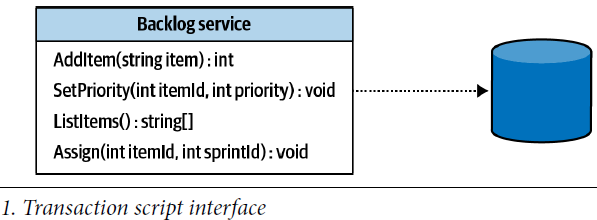
# Transaction Script

Martin Fowler defines this pattern as:

***Organizes business logic by procedures*** *where* ***each procedure handles a single request from the presentation.***

A system’s public interface can be seen as a collection of business transactions that

consumers can execute:



**These transactions can retrieve information managed by the system, modify it, or both.**

The pattern organizes the system’s business logic based on procedures, where each procedure implements an operation that is executed by the system’s consumer via its public interface.

In effect, the system’s public operations are used as encapsulation boundaries.

## Implementation

The only requirement procedures have to fulfill is transactional behavior:

***Each operation should either succeed or fail but can never result in an invalid state.***

Even if executionof a transaction script fails at the most inconvenient moment, the system should**remain consistent—either by rolling back any changes it has made up until the failureor by executing compensating actions.**

The transactional behavior is reflected in thepattern’s name: transaction script.

Here is an example of a transaction script that converts batches of JSON files into

XML files:

**DB.StartTransaction();**

var job = DB.LoadNextJob();

var json = LoadFile(job.Source);

var xml = ConvertJsonToXml(json);

WriteFile(job.Destination, xml.ToString();

DB.MarkJobAsCompleted(job);

**DB.Commit()**

## It’s not that Easy and it’s Important to Know it Well

**the transaction script pattern is a *foundation* for the more advanced business logic implementation patterns you will learn in the forthcoming chapters.**

Furthermore, despite its apparent simplicity, **it is the easiest pattern to get wrong.** A considerable number of production issues I have helped to debug and fix, in one way or another, often boiled down to **a mis-implementation of the transactional behavior of the system’s business logic.**

Let’s look at some of the mistakes and challenges of implementing the transactional behavior:

### Lack of Transactional Behavior

A trivial example of failing to implement transactional behavior is to issue multiple updates without an overarching transaction.(not having a transaction that spans both updates)

If any issue occurs after the first update and before the second one, the system will end up in an inconsistent state.

The issue can be due to anything **from a network outage** to **a database timeout** or **deadlock**, or **even a crash of the server** executing the process.

The problem is easy to fix for a relational database by introducing a transaction encompassing both changes:

try

{

\_db.StartTransaction();

\_db.Execute(@"UPDATE **Users** SET last\_visit=@p1 WHERE user\_id=@p2", visitedOn, userId);

\_db.Execute(@"INSERT INTO **VisitsLog**(user\_id, visit\_date)VALUES(@p1, @p2)", userId, visitedOn);

\_db.Commit();

} catch {

\_db.Rollback();

throw;

}

The fix is easy to implement due to relational databases’ native support of transactions

spanning multiple records.

* **Things get more complicated when you have to issue multiple updates in a database that doesn’t support multi-record transactions**
* **or when you are working with multiple storage mechanisms that are impossible to unite in a distributed transaction.**

Let’s see an example of the latter case:

### Distributed Transactions

In modern distributed systems, it’s a common practice to make changes to the data in

a database and then notify other components of the system about the changes by publishing messages into a message bus.

Consider that in the previous example, instead of logging a visit in a table, we have to publish it to a message bus:

public void Execute(Guid userId, DataTime visitedOn){

\_db.Execute("UPDATE Users SET last\_visit=@p1 WHERE user\_id=@p2",visitedOn,userId);

\_messageBus.Publish("VISITS\_TOPIC",new { UserId = userId, VisitDate = visitedOn });

}}

As in the previous example, any failure occurring after \_db.Execute but before \_messageBus.Publish succeeds will corrupt the system’s state.

The Users table will be updated but the other components won’t be notified as publishing to the message bus has failed.

Unfortunately, fixing the issue is not as easy as in the previous example. **Distributed**

**transactions spanning multiple storage mechanisms are complex, hard to scale, error**

**prone, and therefore are usually avoided.**

In Chapter 8, you will learn how to use **the CQRS architectural pattern to populate multiple storage mechanisms.** In addition, Chapter 9 will introduce **the outbox pattern, which enables reliable publishing of messages after committing changes to another database.**

Let’s see a more intricate example of improper implementation of transactional behavior.

### Implicit Distributed Transactions

Consider the following deceptively simple method:

**public void** Execute(Guid userId)

{

\_db.Execute("UPDATE Users SET visits=visits+1 WHERE user\_id=@p1",userId);

}

Instead of tracking the last visit date as in the previous examples, this method maintains

a counter of visits for each user. Calling the method increases the corresponding

counter’s value by 1. All the method does is update one value, in one table, residing in

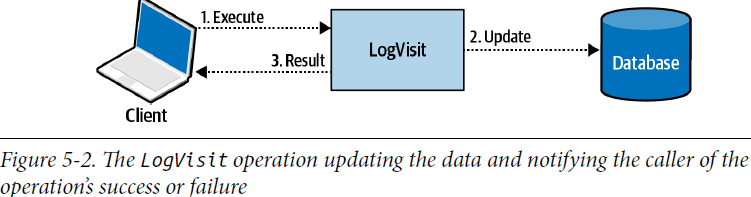
one database. Yet this is still a distributed transaction that can potentially lead to

inconsistent state.

This example constitutes a distributed transaction because it communicates information

to the databases and the external process that called the method, as demonstrated

in Figure 5-2.



Although the execute method is of type void, that is, it doesn’t return any data, it still

communicates whether the operation has succeeded or failed: if it failed, the caller will get an exception.

**What if the method succeeds, but the communication of the result to the caller fails**? For example:

* + If LogVisit is part of a REST service and there is a network outage
  + If both LogVisit and the caller are running in the same process, but the process fails before the caller gets to track successful execution of the LogVisit action?**(this I did not understand)**

**In both cases, the consumer will assume failure and try calling LogVisit again. Executing the LogVisit logic again will result in an incorrect increase of the counter’s**

**value.** Overall, it will be increased by 2 instead of 1.

As in the previous two examples, the code fails to implement the transaction script pattern correctly, and inadvertently leads to corrupting the system’s state.

**As in the previous example, there is no simple fix for this issue.**

It all depends on the business domain and its needs. In this specific example, **one way** to ensure transactional behavior **is to make the operation *idempotent***: that is, leading to the same result even if the operation repeated multiple times.

(I have added some notes about this solution on the next page)

For example, we can ask the consumer to pass the value of the counter. To supply the

Counter’s value, the caller will have to read the current value first, increase it locally,

and then provide the updated value as a parameter. **Even if the operation will be executed multiple times, it won’t change the end result:**

**public class LogVisit**

{

**public void** Execute(Guid userId, **long** visits){

\_db.Execute("UPDATE Users SET visits = @p1 WHERE user\_id=@p2",visits, userId); }}

*I add: here is another problem: the caller reads the value, and by the time it commits the new result another thread has updated the value, one way to address this is to lock the row for the entire transaction where the read operation holds the lock up until the transaction commit so no one else could change the value in between the read and write operations, another way is to use the concept of optimistic locks. Notice that in the following paragraph an optimistic lock is used for the original solution that was not idempotent and only addresses the problem with retries, and not concurrent modifications as if it happens, the first request will not have any effect what so ever:*

Another way to address such an issue is to use **optimistic concurrency control**: prior to calling the LogVisit operation, the caller has read the counter’s current value and passed it to LogVisit as a parameter. LogVisit will update the counter’s value only if it equals the one initially read by the caller:

**public class LogVisit**

{

...

**public void** Execute(Guid userId, **long** expectedVisits){

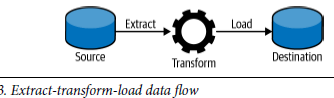
\_db.Execute(@"UPDATE Users SET visits=visits+1 WHERE user\_id=@p1 and visits = @p2", userId, visits);}}

Subsequent executions of LogVisit with the same input parameters won’t change the data, as the WHERE...visits = @prm2 condition won’t be fulfilled.

## When to use Transaction Script?

The transaction script pattern is well **adapted to the most straightforward problem domains in which the business logic resembles simple procedural operations.** For example, **in extract-transform-load (ETL) operations, each operation extracts data from a source, applies transformation logic to convert it into another form, and loads the result into the destination store.**

This process is shown below:



* The transaction script pattern naturally fits supporting subdomains where, by definition, the business logic is simple.
* It can also be used as an adapter for integration with external systems—for example, generic subdomains, or as a part of an anticorruption layer (more on that in Chapter 9).

The main advantage of the transaction script pattern is its simplicity. **It introduces**

**minimal abstractions** and minimizes the overhead both in runtime performance and

in understanding the business logic. **That said, this simplicity is also the pattern’s disadvantage:**

The more complex the business logic gets, the more **it’s prone to duplicate business logic across transactions, and consequently, to result in inconsistent behavior— when the duplicated code goes out of sync.**

**As a result, transaction script should never be used for core subdomains, as this pattern won’t cope with the high complexity of a core subdomain’s business logic.**

This simplicity earned the transaction script a dubious reputation. Sometimes the

pattern is even treated as an antipattern. After all, if complex business logic is implemented

as a transaction script, sooner rather than later it’s going to turn into an unmaintainable, big ball of mud.

It should be noted, however, that **despite the simplicity, the transaction script pattern is ubiquitous in software development. All the business logic implementation patterns that we will discuss in this and the following chapters, in one way or another, are based on the transaction script pattern.**

# Active Record