A cost-flexible approach to transforming a legacy PL/I application to perform an asynchronous remote service call

Author(s):
Ambrosova, Lucia

Publication Date:
2011

Permanent Link:
https://doi.org/10.3929/ethz-a-006450792

Rights / License:
In Copyright - Non-Commercial Use Permitted
Master’s Thesis Nr. 6

Systems Group, Department of Computer Science, ETH Zurich

in collaboration with

Credit Suisse, AG

A cost-flexible approach to transforming a legacy PL/I application to perform an asynchronous remote service call

by

Lucia Ambrošová

Supervised by

Prof. Dr. Gustavo Alonso (ETH Zurich),

Dr. Georg Hüttenegger (Credit Suisse)

October, 2010 - April, 2011
Abstract

This thesis presents a mechanical pattern-based transformation method for introducing an asynchronous communication mode in a legacy PL/I application running on the IMS platform. The method is presented as part of a more generally applicable framework of governing high-level solution concepts. Together, the solutions form a cost-flexible spectrum of holistic approaches ranging from a Synchronous Callout to a full asynchronous Request/Callback-based mode.

The provided reengineering patterns consist of well-defined mechanical steps; hence the resulting cost is highly predictable. As a further benefit, both the patterns and solutions are mutually independent, thereby making the framework modular and in turn facilitating enhancement and replacement on the level of individual components.

Work on this project was initiated as a result of newly arising circumstances in the IT infrastructure of Credit Suisse. For the first time, the mainframe is expected to serve as a client of cross-platform communication and as previous research has shown, synchronous communication may in many cases prove insufficient. The approach demonstrated in this work was designed to mitigate the corresponding deficiencies. It is expected to serve as a foundation for a complete, inexpensive reengineering solution with well-defined risks that will considerably ease the upcoming large-scale migration of legacy applications away from the mainframe.
Acknowledgements

For keeping an eye on me during these 6 months and for his support in times of crisis, I am thankful to professor Gustavo Alonso. I would also like to express my sincere gratitude towards Donat Cornu, Georg Hüttenerger, Rainer Zahradnik and Christoph Gubser and to the rest of the XPP Brokerage team who were always able to find time in their busy schedules to support me both intellectually and technically.

A further thanks goes to Mic Bowman for the initial suggestion that synchronous and fully asynchronous modes might not be the only two alternatives during a workshop in November 2010.

Finally, I would like to reserve a special thanks for my family and for the one who was always there to restore my confidence, motivation and perspective, Mitko.
Declaration of Originality

The work contained in this thesis is original except as acknowledged in the customary manner, and has not been submitted previously for a degree at any university. To the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due credit is given.

The author consents to the thesis being made available for loan and photocopying if accepted for the award of the degree.

_______________________________________

Lucia Ambrošová (lucia.ambrosova@gmail.com)

April 11, 2011
# Table of Contents

1  Introduction ............................................................................................. 1  
   1.1  Motivation ........................................................................................ 1  
       1.1.1  Technical Context ................................................................. 2  
   1.2  Challenges ...................................................................................... 3  
   1.3  Objectives and Outline ................................................................... 3  
   1.4  Prior Work ...................................................................................... 4  

2  Problem Analysis ..................................................................................... 5  
   2.1  Distributed Updates ......................................................................... 6  
       2.1.1  Reservation Pattern ............................................................... 6  
       2.1.2  Compensation Pattern ............................................................ 7  
   2.2  Object Immutability .......................................................................... 8  
       2.2.1  Mutual Exclusion ................................................................. 8  
       2.2.2  Modify Tickets .................................................................... 10  
   2.3  Local Updates .................................................................................. 10  
       2.3.1  Update Postponement ............................................................. 11  
   2.4  State Preservation .......................................................................... 12  
       2.4.1  Data Volume ....................................................................... 12  
       2.4.2  Persistence Level ................................................................. 14  
   2.5  Control Flow Restructuring ............................................................. 15  
       2.5.1  Continuity through Entry Points .......................................... 16  
       2.5.2  Clear Separation ................................................................. 17  
   2.6  Other Problems ............................................................................. 17  
       2.6.1  Testing .............................................................................. 17  
       2.6.2  Reverse Engineering ............................................................... 18  
       2.6.3  Versioning ......................................................................... 19  
       2.6.4  Infrastructure .................................................................... 20  
       2.6.5  Exception Handling ............................................................. 21  

3  Solution Concept ................................................................................... 22  
   3.1  Synchronous callout ......................................................................... 23  
   3.2  Split Transactions ........................................................................... 24  
   3.3  Simple Asynchronous Callout ............................................................ 25  
   3.4  Asynchronous Callout ..................................................................... 26  
   3.5  Summary ....................................................................................... 27  

4  Reengineering Patterns .......................................................................... 28  
   4.1  Launch Remote Notifications ............................................................ 30
Listing 4.8 Organization of offset structures ...................................................... 45
Listing 4.9 Sample pointer tree ........................................................................ 48
Listing 4.10 Sample pointer reallocation for one pointer target ......................... 49
Listing 4.11 Sample RIX tool output ................................................................. 52
Listing 4.12 Sample control flow of a module ................................................... 53
Listing 4.13 Introducing entry points along callout paths ................................ 54
Listing 4.14 Sample callback helper declarations ............................................. 55
Listing 4.15 Introducing entry points on external calls ..................................... 56
Listing 5.1 Callout paths of WC0160 ................................................................. 60
Listing 5.2 EINF_AUFZAEHLEN callback input structure ................................. 61
Listing 9.1 Pre-callout updating code which had to be postponed .................... 72
1 Introduction

When a transformation of a legacy program to perform a remote service invocation is required, is it possible to reduce the negative impact it can have on the throughput and stability of a mainframe application? The naïve answer is yes, it is achievable by invoking the remote service asynchronously, but it comes at a cost. This thesis confirms the positive expectation and demonstrates that the imminent cost can be made flexible, predictable and manageable.

For answers to the impending questions, namely why such transformation would be required; why it would affect crucial properties of the application; why the asynchronous option would remedy the situation; what the risks are; where the additional cost comes from and finally how the cost can be managed, please read on.

1.1 Motivation

Like many other long established institutions, Credit Suisse has in recent years been faced with the challenges brought upon by its possession of a substantial body of legacy IT applications. These very valuable systems have served and grown for a long time and are reaching a stage at which further modifications are no longer viable because the associated costs and risks cannot be justified.

Credit Suisse has taken two mutually supplementary approaches to remedy the situation. First, the legacy application landscape was partially disentangled as part of the DiMa initiative [HC05]. This means that individual applications were modularized, resulting in a higher degree of independence and in turn enabling easier modifications. Secondly, ill-performing mission-critical systems or their components are being replaced by brand new applications. The transformation method introduced in this thesis is expected to be of particular importance in the latter case.

Considering that a lot of time typically passes before an application component has to be substituted by a new application, the replacement is likely to be built on a completely different platform. Hence instead of calling the original local subroutine the encompassing application has to perform a remote call.

This exact situation has surfaced among others in the eXtended Pricing Platform (XPP) Brokerage project, one of the pilots in the Architecture for Core Banking (Arcoba) initiative [RI10]. The goal of this project is to create a client-centric integrated price management system for brokerage products. As an Arcoba pilot, XPP
Brokerage is among the first projects to explore the incompatibilities between the old and the new platform. Its role is to lay down the groundwork on which future project can easily build. This pioneering nature is the reason XPP Brokerage was determined an ideal test bed for the asynchronous callout solution introduced in this thesis.

1.1.1 Technical Context

The goal of XPP Brokerage is to replace the outdated fee and commission calculation application named TARISKA with a new pricing engine [GJ10]. TARISKA is written in PL/I and invoked within a background transaction on the IMS mainframe platform. In contrast, its successor will reside on the state-of-the art Java Application Platform (JAP). Consequently, the surrounding mainframe application has to be reengineered to perform a callout towards the JAP environment.

Application programs on the mainframe run within their assigned IMS regions. These are single-threaded processing units which are dedicated to a single transaction at any given time. Hence while an application idly waits to receive a remote response, it is in fact wasting computing resources. Considering that an external call is expected to take longer than a simple subroutine call due to network overhead, the entire transaction will consume more resources than it used to before the transformation. This is bad news not only for the latency and throughput of the application at hand but clearly also for the overall throughput of applications on the entire machine.

The expected longer transaction runtime carries an additional drawback in terms of database locks. In both IMS and DB2, the two database alternatives used in Credit Suisse, a write lock on a record in fact blocks the whole page. Hence all applications wishing to access the surrounding records have to wait for the current transaction to finish. When the current transaction suddenly takes more time, a real threat emerges of holding up other transactions and by induction causing a chain locking reaction on the mainframe which can in the worst case lead to a complete deadlock.

Needless to say, the potential of adversely affecting unrelated applications has raised question marks over the feasibility of the IMS callout (see [MP10]). The existence of a better solution is thus already turning out to be critical to the success of several ongoing projects within Credit Suisse, for instance EUROM Germany.

Fortunately, since both problems are associated with the extended runtime of the transformed transaction that in turn follows from the long wait period during the
remote call, they can be solved by terminating the transaction during the wait period; in other words by implementing an asynchronous Request/Callback solution.

All technical details in this work apply to IMS background online transactions.

1.2 Challenges

It seems fair to inquire why – if the issue is so pressing – has it not been addressed as soon as the limitations of synchronous callout were known; and more importantly, why the task at hand is considered a difficult one at all.

In XPP Brokerage the answers to these questions are linked through the anticipation that "Changing the communication mode for these remote calls from synchronous to asynchronous would require significant extra re-engineering efforts in the programs calling the component" [KT09]. The threat of having to fund a high and what is worse, an unpredictable cost, was at the heart of the initial decision to implement a synchronous callout. Moreover, the PL/I application that has to be reengineered by XPP Brokerage has multiple clients, all of which may further affect the resulting cost.

Let us examine why reengineering the client application is considered a challenge. Reengineering is typically done in two phases [CE90], namely reverse engineering – deriving an overview of the system mainly from code and documentation – and forward engineering, i.e. manipulating the code to fulfil current requirements. Both phases are highly nontrivial and are in this case further complicated by the high degree of entanglement within the Credit Suisse legacy code [HC05]. The greatest difficulty of reengineering is preserving the original functionality while often making radical changes to the code.

In this case the required changes are indeed drastic. They are primarily driven by the goal of splitting a single transaction into 2 separate local ones while preserving all aspects of the original behaviour. Unsurprisingly, this comes with a full basket of challenges such as application-level business object locking or the need to defer database writes from the first transaction all of which are fully explained later on. Note that the scope of this work covers exclusively changes to the local application.

1.3 Objectives and Outline

The work presented in this thesis aims at defining a cost-flexible mechanical method for the transformation of an internal module call into a full remote asynchronous
invocation. Alternately, the method likewise applies when the initial call is already a synchronous remote callout. The problem is addressed in the following manner:

First, the underlying challenges are analyzed and for each one, multiple solution concepts are developed and compared (section 2). A unified solution is then assembled (section 3) based primarily on the following objectives:

- **Low cost**: Cost is made up of the reverse and forward engineering effort as well as expected maintenance requirements.

- **Efficiency**: To what extent the solution affects the runtime of the program.

Furthermore, on the solution level, general applicability and compatibility with Credit Suisse guidelines are required aspects. Even though most solutions fit in a broad range of settings, they are evaluated with respect to the specific context of IMS.

What emerges is a spectrum ranging from fully synchronous to fully asynchronous invocation alternatives. Each step along the spectrum represents a more complete answer to the synchronous problems but also a more expensive solution. The spectrum also serves as a roadmap because each additional step builds upon the results of its predecessor. The advantage of this approach is its inherent cost-flexibility as each project can select the most appropriate solution on the spectrum.

The thesis continues by formalizing individual work packages in the form of detailed mechanical reengineering patterns (see section 4). Parts of the solution are then verified in a proof of concept implementation (section 5). The paper concludes with a survey of ideas for future research in the subject area (section 6).

### 1.4 Prior Work

Achieving the previously specified goals would have been impossible without the existence of significant prior research in three main topics. The Request/Callback pattern introduced in [CG10] specifies the target solution. Research on reengineering supplies the tools. Among the most prominent here are works on refactoring and reengineering patterns ([MF99], [DS08]). The last big pillar is a synchronous callout prototype [CD10A] which demonstrates the feasibility of making a synchronous call from the legacy mainframe platform towards the JAP and at the same time serves as a basis for the asynchronous callout prototype. A general set of guidelines in the synchronous callout topic is available in [MP10]. Finally, considering that PL/I is no longer a mainstream language, a great value can be attributed to the PL/I references [AP78] and [IBM10].
2 Problem Analysis

As the previous section has suggested, the problem in question comes with numerous challenges. The purpose of this section is to introduce these challenges, focusing mainly on their origin and significance. Additionally, each challenge is addressed with a set of generally applicable solutions. The proposed solutions are either taken from existing literature or, more commonly, developed specifically for this project. Irrespective of their origin, however, all solutions are first explained and their strengths and weaknesses are subsequently discussed in terms of the criteria outlined in Table 2.1 below. Some challenges also give rise to other specific evaluation criteria, which will be introduced in the corresponding subsections. The solutions also include brief overviews of the entailed engineering effort with respect to the specific setting of PL/I applications running on IMS.

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Value range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effort</td>
<td>Low/Medium/High</td>
<td>The expected effort required for implementation. Low implies that the process can be automated.</td>
</tr>
<tr>
<td>Knowledge</td>
<td>Superficial/Detailed</td>
<td>The level of knowledge of the client system necessary for successful implementation. Entails the required amount of reverse engineering.</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Good/Mediocre/Poor</td>
<td>Indicates whether the solution significantly affects the speed of the resulting application.</td>
</tr>
<tr>
<td>Maintenance</td>
<td>Automatic/Manual</td>
<td>Indicates whether the solution requires human involvement to ensure continuous functioning.</td>
</tr>
</tbody>
</table>

Table 2.1 Basic solution evaluation criteria: values are colour-coded with green the most desirable and red the least desirable value.

Most challenges are derived specifically from the need to split the original transaction in order to achieve the target Request/Callback pattern outlined in Figure 2.1. What was originally one physical transaction has to be transformed into three individually committed transactions which together logically form a whole with identical behaviour to the original program. The target pattern also dictates that the two local transactions are highly interdependent. Instead of using a properly managed interface, they are allowed to communicate via a shared state. One may notice that this crucial allowance acts as a double-edged sword where on one hand it facilitates simple solutions to many challenges but is also a source of several new challenges.

In addition to the transaction, the underlying local process has to be freed as well. The goal is to allow processing of other messages during the interlude period. This requirement is necessitated by the single-threaded nature of IMS processing regions.
Naturally, the resulting higher throughput is likewise attainable only through solving a set of associated challenges.

**Figure 2.1 Request/Callback scheme**

### 2.1 Distributed Updates

As soon as remote calls come into play, the first and foremost interest is to ensure data consistency between the local and remote sites. The problem stems from the possibility of failures in the epilogue transaction. In the event that a remote site successfully executes an update operation but the local transaction is forced to rollback, the entire system would be left in an inconsistent state. This is because it violates the desired all-or-nothing commitment semantics which are locally guaranteed by a transaction manager. Since the use of distributed transactions spanning multiple platforms is strictly disallowed within Credit Suisse [GA08], consistency has to be ensured in a different manner.

Credit Suisse guidelines [CG10] allow two distinct techniques to achieve this goal. They are the reservation and compensation patterns explained below and summarized in Table 2.2 at the end of this section. While transformations of remote services are not in scope of this paper, the patterns are mentioned because introducing them also entails making changes to the client application.

#### 2.1.1 Reservation Pattern

The reservation pattern, also referred to as pre-booking pattern, proceeds in two phases as depicted in Figure 2.2. At first, as the name suggests, a remote update is reserved and conditioned with an attached expiry date. Given that no confirmation arrives, the update is reverted – the underlying assumption being that the calling transaction terminated incorrectly. If all goes well, however, a confirmation arrives
and the update is finalized [GA08]. Note that the pattern relies on guaranteed message delivery. It also introduces mechanisms to handle a late confirmation.

Figure 2.2 Reservation pattern swim lane scheme; Source: [FD09]

Clearly, as opposed to the prior case of calling a subroutine within the original transaction, the client now obtains an additional responsibility of sending the confirmation message. On the bright side, this is a largely straightforward task. Let us examine what it consists of.

There is only one important engineering exercise and that is to create a procedure to send the confirmation message and then invoking this procedure when the last transaction executes successfully. Fortunately, the task does not require an in-depth insight into the existing application, so it is possible to capture the issue with a generic set of instructions. Moreover, since the confirmation is a fire-and-forget message [GA08], it does not impede the runtime of the application. Finally, the solution does not have any special maintenance claims because it does not break in reaction to unrelated changes in the application.

2.1.2 Compensation Pattern

In direct contrast to the reservation concept is the idea behind the compensation pattern. Here, an 'undo' transaction is defined with the purpose to revert any changes done by a previously committed transaction. This solution is used seldom as it is tricky to implement and error prone because it may lead to unpredictable side-effects. However, it is a valid option and is therefore considered here.
Regardless of conceptual differences, the two patterns are almost equivalent from the perspective of the client application. A follow-up message still has to be sent, only instead of a confirmation message on successful commit, the application uses it to request an undo operation in case anything goes wrong. These semantic differences play no role in terms of reengineering requirements (see Table 2.2).

<table>
<thead>
<tr>
<th></th>
<th>Effort</th>
<th>Knowledge</th>
<th>Efficiency</th>
<th>Maintenance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reservation</td>
<td>Medium</td>
<td>Superficial</td>
<td>Good</td>
<td>Automatic</td>
</tr>
<tr>
<td>Compensation</td>
<td>Medium</td>
<td>Superficial</td>
<td>Good</td>
<td>Automatic</td>
</tr>
</tbody>
</table>

Table 2.2 Local transformations for distributed updates: solution overview

2.2 Object Immutability

The previous section tackled the problem of absence of typical transactional properties – atomicity and consistency – in the context of a distributed system. At its essence, the next challenge is no different. This time it concerns the "I" in the well-known ACID properties. Isolation turns out to be yet another property that fails to transcend the boundaries of a single physical transaction.

The role of isolation is to prevent concurrently running transactions from accessing each others' uncommitted modifications. By implication, it also prevents the scenario of concurrent transactions modifying the same data fields. However, the step from a synchronous to an asynchronous solution requires splitting the local physical transaction. Hence, there now is a period of time when all objects are freely modifiable by other processes despite the fact that the local transaction is logically still running. In other terms, isolation is potentially violated.

The goal is thus to restore isolation and in turn data integrity across several consecutive physical transactions. The idea behind the proposed solutions is to assure data immutability through application-level code. The other side of the coin, namely disallowing other transactions to read physically committed modifications before the logical transaction is successfully finalized, is discussed in section 2.3.

2.2.1 Mutual Exclusion

Mutual exclusion has been a well-studied principle since the dawn of databases. A transaction implicitly or explicitly acquires a lock on a resource, such as a record in a database, in order to prevent other transactions from overwriting its uncommitted changes. The IMS databases rely on explicit locking. More specifically, immediately
before updating an object, each transaction has to issue a blocking "Get Hold Unique" (GHU) command. By doing so, it secures an exclusive write privilege on this object and IMS takes over ensuring isolation from other writes.

One of the most critical factors in the challenge of extending mutual exclusion to multiple consecutive transactions is transparency. Having to implement the solution on each database client separately is simply not an option due to the associated costs and liabilities. Yet it is vital that each application accessing these objects respects the extended locking mechanism. To this end, the solution is to exploit the existing GHU interface. It is worth noting that mainframe applications typically access a database through a shared application-level module that will be further referred to as a driver. Modifying the driver's GHU procedure to be able to secure the extended locks is hence the key to achieving transparency. Importantly though, modifying such fundamental functionality has to be done with extra care. Databases employ fine-tuned deadlock recognition and avoidance techniques. Breaking these could ultimately create room for new deadlocks in all of the database client applications.

The extended lock is in its essence a key plus an expiring stamp on a record. If these fields are set and have not expired, the record can only be handled by the key holding transaction. In other words, the GHU implementation blocks until the record either does not have a lock, or the lock has expired, or the transaction can provide the key. Moreover, the driver's REPL (update) method needs to be reengineered to release the current lock. At the same time, the expiry policy guarantees that the object is accessible even if the lock is not lifted by the key holding transaction. The final step is to introduce the extended lock setting method, which calls GHU and sets the key and expiry fields. This method does not have to be called by applications that do not require the extended lock because they are covered by the physical lock. It is sufficient for them to use the extended GHU mechanism and subsequently perform the desired update.

In case of DB2 databases, which have no explicit locks, a record is locked implicitly upon an update operation and other updates are blocked until the lock is lifted. Hence, a similar solution can still be implemented – where instead of checking the extended lock upon GHU, it is checked whenever an updating call is made.

Achieving mutual exclusion with privileges lasting through several transactions is a relatively simple exercise in reengineering terms. Nevertheless, as was pointed out, this solution might suffer from deadlock issues and thus requires thorough testing.
2.2.2 Modify Tickets

The second solution is derived from the well-known concept of optimistic concurrency control [KH81]. Rather than preventing other transactions from updating an object, the modify ticket approach is to rely on the object to not be touched and verifying this fact before proceeding with local modifications [TK08]. As it does not impose changes on other database clients, such non-blocking approach to isolation results in a more light-weight solution. There are two ways to deal with a negative response from a modify ticket verification. The first option is to terminate the transaction and start over. The downside of this choice is not only the many new reengineering challenges it produces, but more importantly its inherent tendency to waste time and computing cycles. With these deficiencies in mind, the use of modify tickets is restricted to specific conditions for the purposes of this paper, namely to records which are expected not to be accessed concurrently. The expectation allows to treat a modify ticket failure as a technical failure that leads to termination of processing.

The modify ticket is commonly implemented using a hash value derived from the initial read of an object. Obtaining a different hash value from the currently persisted object before executing an update gives away an unwanted foreign update. In order to engineer this solution, the corresponding driver has to be modified to compute a hash value of an object on every read. Introducing a validation step in the callback with a fast track to termination concludes the necessary reengineering efforts.

All in all, the solution is only suitable for sequentially accessed objects, a good example being objects processed by serial workflows. However, within these restrictions, it is a clean, easily achievable solution, which does not create any deadlock threats and is already well-established in Credit Suisse [HC05].

<table>
<thead>
<tr>
<th>Exclusion</th>
<th>Effort</th>
<th>Knowledge</th>
<th>Efficiency</th>
<th>Maintenance</th>
<th>Pitfalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modify tickets</td>
<td>Medium</td>
<td>Superficial</td>
<td>Good</td>
<td>Automatic</td>
<td>Restricted use</td>
</tr>
</tbody>
</table>

Table 2.3 Object immutability: solution overview

2.3 Local Updates

Section 2.1 addressed the typical problems associated with data consistency in distributed systems. The discussion now has to be taken one step further as the problems reappear locally due to the targeted transaction split. The new challenge is to assure the all-or-nothing semantics also for the logical local transaction which now
Problem Analysis

consists of two physical ones. The rule of thumb that is applied in this section is: "Move all definitive database updates to the callback". Since the callback ultimately decides whether the transaction was successful, it needs to be able to control all resulting operations.

The two approaches we have seen in the remote context are the reservation and compensation patterns. Technically, the updates in both solutions are not definitive. In the first case they have to be confirmed to become effective and in the latter case updates can simply be undone. However, in the local environment, committing an update as suggested by the compensation pattern is not acceptable. Even if the provided eventual consistency would be sufficient, by committing what is logically a partial update, the overall transaction would break isolation (see section 2.2).

Apart from the reservation pattern, the local setting permits another reengineering solution, namely deferring the actual database update operations until the callback. In terms of reengineering, the options are in fact similar. Regardless of the intentions, both solutions entail changing the updating modules deep down the call hierarchy. At the same time, given that the state of the program is persisted (see section 2.4), the two alternatives effectively lead to the same behavior. However, the reservation pattern makes an implicit claim to be recognized by other programs, resulting in a load of additional engineering requirements. Consequently, the reservation pattern will not be considered.

2.3.1 Update Postponement

Before it can update an object, a program has to first read this object from its corresponding storage location. It is likely that other than for the purpose of updating the object's value in the storage location, a program may use the object to determine a further execution strategy. Therefore, even though the goal is to postpone the updating operations, the associated reads have to be left in place. Note that in comparison, the updates generally do not affect the further flow of the program other than by causing immediate technical failures. Hence, they can be freely moved into later parts of the program.

In order to postpone an update, one therefore only has to determine which parts of the code affect the program and which are only used for the purpose of updating the object's value. Based on this knowledge, the code can then be separated into distinct reading and updating modes. The updates are then executed in the last callback. The engineering effort is therefore relatively low. On the other hand, such a
postponement technique requires an overview of the possible updates. Because the updates can be performed in a large number the program's dependencies, the required overview is not only difficult to gain (see section 2.6.2) but also hard to keep up-to-date later on. Hence, the only available solution to the problem of keeping local transactions in sync imposes significant costs in terms of both the reverse engineering and maintenance factors.

In IMS databases, a program is allowed to update only if the GHU read has been performed in the same transaction. It follows that the program has to be engineered to repeat the read in the callback, thus having slightly detrimental effects also on the overall efficiency of the application.

<table>
<thead>
<tr>
<th>Postponement</th>
<th>Effort</th>
<th>Knowledge</th>
<th>Efficiency</th>
<th>Maintenance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medium</td>
<td>Detailed</td>
<td>Mediocre</td>
<td>Manual</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4 Local updates: solution overview

2.4 State Preservation

Another challenge associated this time with the need to free the local process is the need to preserve the state of the interrupted program. The state is made up of application-relevant data as well as control variables (e.g. loop counters). The goal is for each subsequent transaction to be able to take over where its predecessor left off.

The proposed solutions exist in a two-dimensional space defined by two axes: the amount of data and the way the data is preserved. Points on both axes range from working yet suboptimal solutions to their more complete and consequently more costly counterparts. The following subsections present the individual axes and explore the corresponding trade-offs in terms of the standard evaluation criteria.

2.4.1 Data Volume

The first axis concerns the amount of data that is retained between the two local transactions. Regardless of how the data is stored during this time, in the light of resource conservation and maintainability it appears ideal to minimize this amount. Let us examine the sacrifices that have to be made on the road towards this goal.

The elementary solution is to regard the state of a program to be made of all of its variables – on all levels of the call stack. Admittedly, volume-vise this is a downright suboptimal proposal. Among other redundancies, such state would likely include variables that are entirely unused in the follow-up transaction. On the other hand,
the inherent redundancy of the state enables carefree modifications to the program without requiring changes to the state. For instance, modifying a follow-up transaction to use a previously unused variable from the preceding transaction comes without any trouble. Due to the good practice of explicit variable declaration, it is also trivial to determine the full state. However, the state may consist of pointers which can be dynamically allocated in any of the program’s submodules. Coupled with the IMS-specific possibility of declaring pointers without specific types, finding the structure of the full state becomes a significant reverse engineering challenge. Furthermore, the knowledge has to be kept up-to-date, incurring maintenance costs.

The first step towards reducing data volume is preserving only dirty data. These are structures that have been altered by the program or any of its submodules. Where does the remaining data come from? The remaining information is read-only, which means that if needed, it can simply be retrieved again in the callback. Although it has not been investigated what fraction of data is typically read-only, based on an initial inspection of the XPP code the assumption is that the resulting data volume reduction would be substantial. The bad news is that accomplishing this solution is not easy. The reverse engineering part, i.e. finding which variables are modified, means trivial but tedious code analysis and it suffers from analogous pointer-related problems as the previous solution. Additionally, engineering the repeated reads requires care. This is because database reads are typically performed by a variety of shared and often complex modules. It has to be ensured that calling these modules repeatedly will not produce any undesirable side-effects - a task, that is difficult on both reengineering fronts. To top it all, changing the program to meet needs such as newly accessing a read-only variable in the callback, may solicit further reengineering requirements.

<table>
<thead>
<tr>
<th></th>
<th>Effort</th>
<th>Knowledge</th>
<th>Efficiency</th>
<th>Maintenance</th>
<th>Data Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>All data</td>
<td>Low</td>
<td>Detailed</td>
<td>Good</td>
<td>Manual</td>
<td>Large</td>
</tr>
<tr>
<td>Dirty data</td>
<td>Medium</td>
<td>Detailed</td>
<td>Mediocre</td>
<td>Manual</td>
<td>Moderate</td>
</tr>
<tr>
<td>No data</td>
<td>High</td>
<td>Detailed</td>
<td>Poor</td>
<td>Manual</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 2.5 Preserved data volume: solution overview

The most radical solution is to completely remove the necessity to preserve data. Even though it eradicates the pointer-related trouble, this option is no exception to the trend of increasing difficulty. Apart from the problems associated with the previous solution, this one would also entail storing the partial progress on all data into their respective databases. However, as suggested in [CG10], if this was possible
Problem Analysis

with reasonable engineering effort and risk, the Request/Callback pattern that is at the heart of this paper would be obsolete.

2.4.2 Persistence Level

The second factor of data preservation concerns the form in which information is stored during the interlude. The points along this axis differ in the levels of persistence they provide. In other words, the way in which they enable the logical transaction to continue after the local platform goes through a crash and recovery. Recall that among the prime requirements of this project is conservation of the functionality of the original application. Consequently the recovery process of the new logical transaction has to emulate the behavior of the original single physical transaction. A semantic equivalent of a full rollback and a restart is thus required.

Figure 2.3 designates the 3 time ranges to be considered for a recovery analysis. A crash during the prologue automatically leads to a clean restart, which is in line with the target behavior. The remaining two potential crash points, however, limit the space of feasible solutions.

![Figure 2.3 Crash possibilities throughout the logical transaction](image)

The fact that all transactions of one type generally run within the same mainframe area means that they can share memory among each other. Hence, the simplest way to retain the state of a program is to hold the data in shared memory. As before, the charm of a simple solution lies in the ease of implementation. But just like before, there is a drawback. In this case, recovery problems are brought about by the ingrained lack of persistence. More specifically, a system crash during the latter two phases would lead to a data loss and would make it impossible to (re-)start the epilogue. The target behavior is then not met as the transaction will not be automatically restarted.
Taking retention to the next level means persisting the state in a database. It does not demand significantly higher effort. Using appropriate tools, the mapping to a database can be generated automatically. While it solves the recovery problem, what makes the proposal unfavorable is the inflexibility of an underlying database schema. Inevitably, programmers wishing to change the structure of a program's state would be forced to get involved in the technicalities of this solution.

Dynamic mapping to text or other representations offers itself as mitigation to the above mentioned problem. The data can be stored in a file or in a database as a BLOB. The downside of dynamic behavior can be decreased efficiency.

<table>
<thead>
<tr>
<th></th>
<th>Effort</th>
<th>Knowledge</th>
<th>Efficiency</th>
<th>Maintenance</th>
<th>Recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-memory</td>
<td>Low</td>
<td>Superficial</td>
<td>Good</td>
<td>Automatic</td>
<td>No</td>
</tr>
<tr>
<td>Unstructured</td>
<td>Low</td>
<td>Superficial</td>
<td>Mediocre</td>
<td>Automatic</td>
<td>Yes</td>
</tr>
<tr>
<td>Structured</td>
<td>Low</td>
<td>Superficial</td>
<td>Good</td>
<td>Manual</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2.6 State persistence levels: solution overview

### 2.5 Control Flow Restructuring

When it comes to splitting the local program in two, the task with arguably the most visible result is to restructure the modules involved in the callout. These changes determine how the code will be read by developers in the future. The challenge is therefore not a technical one but rather of cognitive nature. An ideal outcome is for the code to be easily understandable and clean at the same time.

The solution concepts developed for this problem are based on a decision to keep both the prologue and epilogue code in one module. The reasons are twofold. Firstly, the two parts belong to one logical program. Secondly, having originally formed a single program, they are likely to use a common set of internal procedures. The two subprograms may, however, run under two different transactions, each having a distinct and identifiable path in the code.

What remains is to address the problem of multiple code paths towards the callout. The callback needs to finish the computation in correspondence to the original flow. This problem is essentially a generalization of a simpler situation, more specifically the possibility of multiple different calls that may trigger the callout occurring in a single procedure. Therefore, the solution concepts focus on the analysis of this low-level situation.
2.5.1 Continuity through Entry Points

The first solution emphasizes continuity. The idea is for the program's control flow to be structured as though the program itself was not interrupted, thereby facilitating natural reading and understanding. As an exception to the rule of general applicability of solutions, the approach here is to exploit the PL/I-specific entry construct. The underlying notion allows a procedure to be entered at multiple labeled locations. It is therefore possible for the callback to enter a procedure at a point immediately after the callout location.

Listing 2.1 below demonstrates the basic reengineering approach. The prologue is allowed to run as before, with the exception that a corresponding entry point for the callback is saved before each call to a new procedure or module that eventually triggers an asynchronous callout. After the callout is triggered, each procedure on the call stack immediately terminates. At this point, the prologue is over, having left behind a set of entry points that the epilogue will use to rebuild the call stack. Once that is done, the execution can continue as though there was no interruption.

```plaintext
procedure main:
  ...commands before...;
  call saveMainEntry(main-entry-x); // save callback entry
  call procedureA(); // procedure that initiates the callout
  return; // terminate the program after callout
  main-entry-x: entry; // callback enters here to continue
  call getProcedureAEntry(); // finish off procedureA
  ...commands after...;
main-end;
```

**Listing 2.1 Demonstration of in-code entry points.** Original code is in black, blue highlights newly added lines.

Identifying the code paths towards the callout and adding a few generic lines of code around each forwarding point along the paths is all it takes to implement this solution. Clearly, the task requires no expert knowledge and can be fully automated. Additionally, it has no impact on the efficiency of the application and maintaining it translates to adding 5 lines of code instead of 1 whenever a new call on the path is introduced. To summarize, the solution is highly satisfactory in terms of the standard evaluation criteria. It promotes code understanding through retaining continuity. However, the reengineering process may be detrimental to code quality as a result of packing the code full of entry points and return statements.
2.5.2 Clear Separation

An emphasis on the execution disconnect that gives rise to the next solution is in direct contrast with the continuity concept. A clear separation of the prologue from the epilogue code aims to make the developers aware of the interruption at first glance. The challenge is to simultaneously enable them to follow the flow of the logical program as easily as possible.

While splitting a procedure that contains only a single callout path is a simple task, complications arise whenever there are multiple possible asynchronous invocations. First, for the sake of readability, the prologue has to be reengineered to contain a minimum number of callout-triggering statements. This can be achieved by merging relevant identical statements. Second, the callback code has to be separated and logically linked to the prologue statements. Due to the unavoidably high variability in different program implementations, it is not trivial to specify a streamlined transformation method. This task is so code-dependent that human involvement is absolutely vital. After all, code structure and quality is to a large extent a cognitive factor, which means it is only maintained by humans for other humans.

Overall, this solution sets forth a considerably different set of objectives than the previous continuity-oriented approach. Here, clear separation and code quality take the lead, leaving behind the comfort of automation (see Table 2.7). Moreover, as the restructured code is readable, maintenance should not pose a problem.

<table>
<thead>
<tr>
<th>Continuity</th>
<th>Effort</th>
<th>Knowledge</th>
<th>Efficiency</th>
<th>Maintenance</th>
<th>Code quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separation</td>
<td>Medium</td>
<td>Detailed</td>
<td>Good</td>
<td>Automatic</td>
<td>Preserved</td>
</tr>
</tbody>
</table>

Table 2.7 Control flow restructuring: solution overview

2.6 Other Problems

This section provides an insight into other identified challenges that are associated with the problem at hand. The precise answers to these challenges are out of scope of this thesis. However, some hints and high-level solution proposals are included.

2.6.1 Testing

The principal requirement of the reengineering project at hand is to not alter the externally visible behavior of the application. The most reliable indicator of success is
a proper set of testing instruments. As established by the authoritative work on reengineering patterns [DS08], tests are a life insurance. The book provides a full set of testing patterns regarding 'when', 'why', 'how' and 'what' to test. The following paragraphs elaborate on the 'what' part with regard to the asynchronous callout task.

Behavior equivalence of the original application and the reengineered version comes in many tastes and flavors. The programs have to produce an identical set of updates for all use cases. This means in all possible execution flows, but also in case of any exceptions. The first goal is therefore to create a comprehensive test body for side-by-side behavioral comparison that can serve as a regression test.

Throughout this problem analysis chapter, solutions have been proposed that could profoundly affect the relevant application or even a broader set of programs. Two examples of such offenders are the locking mechanism (section 2.2.1) and control flow refactoring (2.5.2) solutions. Since their pitfalls are well defined, it makes sense to test them specifically rather than expecting them to be captured by the regression tests. These fine-grained tests represent the second important test instrument.

In contrast with the desired behavioral consistency, the nonfunctional expectations are quite disparate. While introducing a remote call will naturally deteriorate the runtime of individual programs, the overall throughput is expected to improve. The asynchronous solution is additionally presumed to manifest a degree of operational independence from the remote service. The third class of tests should therefore be used to verify and gauge the progress of these nonfunctional factors.

Finally, many IT applications in Credit Suisse have a set of dedicated regression tests – often testing a full workflow rather than just a single program. While they are unsuitable to test the numerous subtleties of the reengineering solution, their benefit is that they likely test the most typical types of workloads. If passed, this is just an additional assurance that the solution is working.

The four above mentioned test instruments together form a comprehensive package. Determining the appropriate test volume is about balancing effort and risk.

### 2.6.2 Reverse Engineering

As one of the two components of any reengineering exercise, reverse engineering plays a supporting role in the process. Yet in terms of the challenges it brings on the table, it does not get overshadowed by its forward counterpart. The level of reverse
engineering required for each task has been analyzed and showed up in the solution overview tables under the title Knowledge. Some of the major proposals that are not achievable without detailed knowledge of the system are listed here.

Update postponement in section 2.3.1 was the first of these tasks. Due to the entanglement on the host platform, identifying all updates that occur prior to the remote call is a highly nontrivial task. A quick code hierarchy review suffices to establish the complete collection of modules that may be referenced by the application. The total set of update operations performed by these modules is however a gross overestimate of the true subset that is in fact used by the application. If one existed for PL/I, using a static call graph tool would be a good option, but the resulting set would still likely be too large. Finding the true updates means drilling down to details in order to discover which updates are reachable from the main application. Overall, the task is highly demanding, but can be made easier through some shortcuts. For example, large portions of updating modules can be freely disregarded because they are only referenced in the callback.

The next reverse engineering-heavy task is to analyze database "reading" modules for idempotence as required by the data volume reduction technique from section 2.4.1. The goal is to see if the service can be called twice, in other words whether it causes any undesired side effects. However, these side-effects could be so diverse that the real challenge rests already in the lack of a clear understanding of what should be looked for.

Procedure separation from section 2.5.2 brings in essence an even more undefined problem. The task is to gain familiarity with one procedure at a time in order to be able to split it in two. Even though the task is difficult to formulate, there is some general advice, for example to look for repeating callout-related code segments that can be later grouped.

**2.6.3 Versioning**

In the Request/Callback pattern the local transactions are logically part of one program, which means they have dependencies. These are captured in the data that makes up the shared state. Hence, if the state structure changes, it is necessary to ensure that all messages are processed by an epilogue running the same version as the prologue. At any given point there are messages in flight between the two parts. The question is therefore how to upgrade the modules while making sure that all currently in-flight messages are still processed by the old version.
A possible solution is to freeze the input message queue and wait until all in-flight messages have passed through the epilogue before upgrading the modules and opening up the queue again. However, the asynchronous communication mode implies loose coupling, which means no assumptions can be made about the duration of the callout. The upgrading procedure may therefore technically take an arbitrary amount of time, during which time the program would effectively not work.

The outlined problem can be remedied if both the old and new epilogue versions can process the corresponding messages simultaneously. To this end one needs to design a mechanism that would multiplex the remote response messages to the correct queue based on the desired version number. This implies having two different versions of each module temporarily running in parallel. The solution hence requires that all old modules on the dependency hierarchy are renamed for this time period, which is a slightly tedious refactoring exercise.

The final alternative is to account for each change directly in the code. The version number would have to be passed to all parts of the programs which rely on newly introduced state variables and this new code would be executed conditionally based on the version. Additionally, these conditions would have to be removed once all old-version messages have been processed in order to avoid cluttering the code. This choice therefore requires a lot of manual engineering.

2.6.4 Infrastructure

The Request/Callback pattern allows several retries of the remote call in case of technical failures. The target infrastructure therefore has to support this option. At the same time, in order to shield the application from the associated technical difficulties, the complexity should be offset to the infrastructure. The epilogue application should thus only be invoked only once the final response is known.

The proposed infrastructure in Figure 2.4 is based on the infrastructure of the existing PoC of synchronous callout which is included as an appendix in Figure 9.1. The Service stub, Proxy and Asynchronous callout can be obtained from the artifacts generated by IFMS (Interface Management System) as described in [CD10A]. Storage manager, Callback driver and Message envelope reader do not depend on the specifics of the application and can therefore be standardized. Due to a lack of standard marshalling mechanism on the mainframe, the Service input/output adapters have to be created manually.
Figure 2.4 Target infrastructure of the asynchronous callout. Green colour shows existing application modules on the callout path.

The Call database stores a storage identifier generated by the storage module. It also stores the remote request message which can then be reused by the Callback driver in case of technical failures. For simple identifications, failure notifications should be indicated in the envelope of the response message. Finally, the database also contains all response messages, the most recent of which is then unmarshalled and returned when the callout module invokes serviceResponse. This element of the interface of Service stub has to be created from scratch as it is not a standard artifact generated by IFMS. All queues should be persistent.

2.6.5 Exception Handling

Most solutions introduced in this chapter introduce possibilities for new error situations. For instance, unlike the original local subroutine call, the remote callout can fail due to technical reasons such as queues being full. Another example is a modify ticket comparison failure. While the handling of communication failures in the Request/Callback pattern is described in [CG10], other newly introduced exceptional cases remain to be worked out in parallel with all solutions in the future.


3 Solution Concept

The purpose of this paper is to introduce a complete plan for reengineering a subroutine call into a remote asynchronous request. As shown in the previous chapter, there are many challenges associated with this problem. Furthermore, most challenges do not have a single perfect answer, but rather a set of competing solutions – each offering different trade-offs and none being a clear winner. The solutions were evaluated along many dimensions, such as the amount of engineering effort, maintenance or efficiency of the resulting application, but also functional aspects like crash recovery potential. How can one assemble the most appropriate collection of solutions from this high-dimensional space into a coherent approach? That is the question addressed in this chapter.

The key observation is that once again there might not be a single best answer. Projects may have a large variety of goals and preferences. Therefore, instead of playing the one-size-fits-all game, it makes sense to combine solutions with collective trade-offs weighing towards a specific goal. The most ubiquitous driving force behind design decisions is the associated cost. Regardless of other specific requirements, every project is interested in delivering the most cost-effective solution. Thus, cost makes for a suitable criterion to consider when assembling final solutions.

The total cost of a transformation to an asynchronous callout is determined by the amount of reengineering effort required. This includes the reverse and forward engineering effort as well as maintenance claims, i.e. entries in the solution overview tables (see Table 2.1) appearing under 'knowledge', 'effort' and 'maintenance', respectively. Unsurprisingly, one can observe an inverse relationship between the cost generating values and all other attributes. In other words, quality is expensive.

What is then the right balance? The answer is fundamentally project-dependent. Let us instead ask the right question: how can we empower each project to pursue the most appropriately balanced solution? Figure 3.1 below demonstrates the core idea of this thesis. It shows a spectrum of approaches, starting at a local subroutine and leading to a fully asynchronous solution. Each step along the way represents a more expensive and more complete solution than its predecessor. What is more, an implementation of each step relies on a successful completion of the predecessor. Hence, the spectrum also serves as a roadmap towards an asynchronous callout. It is completely up to the project to choose where to get off the road and accept the corresponding trade-offs. The biggest advantage of the spectrum is therefore that
choosing a more affordable option does not exclude but rather prepares ground for implementing a more complete solution in the future.

---

**Local subroutine**
- develop a test framework

**Synchronous callout**
- handle distributed updates
- generate call infrastructure

**Split transactions**
- assure immutability of essential objects
- postpone local updates

**Simple async. callout**
- add callback infrastructure
- move data to a shared memory area
- generate entry points

**Asynchronous callout**
- refactor control flow on consumer
- persist exclusively dirty data

---

**Figure 3.1 Cost-flexible spectrum of solutions.** Stages are explained on the left, the corresponding units of work are summarized on the right.

An important lesson from Figure 3.1 is that the groundwork for all reengineering efforts is to create a set of regression tests. Furthermore, to cover all bases, the test framework has to be extended with every new step to account for newly added features and track progress in the non-functional directions (see section 2.6).

The individual solutions are examined below in terms of effort and drawbacks. A table symbolically summarizing all evaluation criteria of the combined solutions is presented at the end of each section. The columns in these tables are visually grouped into cost factors, other criteria and finally the two critical aspects, namely whether the solutions cause resource blocking and chain lock reactions.

### 3.1 Synchronous callout

The fastest way to implement a remote service invocation is to do it synchronously. What it takes is to handle remote updates if necessary and create the callout infrastructure. As pointed out in section 2.1, the remote update solutions are comparable from the local perspective – they are both easy to implement. Moreover, the infrastructure can be fully generated from the remote service description. Therefore, the overall cost of this solution is truly minimal.
Unfortunately, the approach bears substantial hazards, all of which were listed in the introductory sections. To reiterate, the blocked transaction not only wastes computing resources on the mainframe, but more importantly, due to the page-wide locking policy on IMS databases, it runs the risk of causing a chain reaction that can lock out many unrelated applications. Consequently, the solution should be used with extra care, especially if the call roundtrip is expected to take considerably longer than the execution of the original subroutine. Additional safety precautions are listed in the dedicated document [MP10]. Under many circumstances the document advises against implementing this solution.

<table>
<thead>
<tr>
<th>distributed updates</th>
<th>Effort</th>
<th>Knowledge</th>
<th>Maintenance</th>
<th>Efficiency</th>
<th>Locking pitfalls</th>
<th>Data volume</th>
<th>Recovery</th>
<th>Code quality</th>
<th>Resource blocking</th>
<th>Chain lock reactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronous callout</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✔️</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3.1 Synchronous callout solution evaluation; Legend: ✔️ - desirable, ✔️ - satisfactory, ✗ - undesirable

3.2 Split Transactions

Both problems encountered in the synchronous callout appear because a transaction and the associated process are blocked throughout the duration of the remote call. To reduce the more severe of these problems, i.e. the threat of locking out other programs, it suffices to assure that no physical locks are held during the callout.

Let us examine which reengineering initiatives can assist this goal. In general, locks are acquired when executing updates. Therefore, updates occurring prior to the callout have to be deferred until after the response comes back as described in section 2.3.1. Some locks may also be acquired implicitly during a read operation that is expected to be followed up by an update. These physical locks have been secured with the sole intention to prevent other applications from modifying the object. Therefore, they have to be replaced by application-level locks (see section 2.2). Finally, if the chosen application-level locking mechanism creates updates, these have to be committed before the callout – by calling a checkpoint operation.

Table 3.2 clearly demonstrates the substantial costs that are incurred by the newly added reengineering projects. Alas, there is no moving on from synchronous callout without performing these changes. On a positive note, progressing to this stage means the subsequent steps are well within the reach. Consequently, even though it
is a good self-contained solution, the prediction is that its biggest purpose will be to serve as a stepping stone towards further solutions.

Table 3.2 Split transactions solution evaluation; Legend: ● - desirable, □ - satisfactory, × - undesirable

<table>
<thead>
<tr>
<th></th>
<th>Effort</th>
<th>Knowledge</th>
<th>Maintenance</th>
<th>Efficiency</th>
<th>Locking pitfalls</th>
<th>Data volume</th>
<th>Recovery</th>
<th>Code quality</th>
<th>Resource blocking</th>
<th>Chain lock reactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>distributed updates</td>
<td>□</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>object immutability</td>
<td>□</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>×</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>local updates</td>
<td>□</td>
<td>×</td>
<td>×</td>
<td>●</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>split transactions</td>
<td>□</td>
<td>×</td>
<td>×</td>
<td>●</td>
<td>×</td>
<td>●</td>
<td>●</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

Table 3.3 Simple asynch. callout solution evaluation; Legend: ● - desirable, □ - satisfactory, × - undesirable

3.3 Simple Asynchronous Callout

The previous step helped to reduce a big threat by splitting the transactions. Now the ground is prepared for removing the remaining obstacle, process blocking. The conceptually simple act of stopping the program after the remote call has been issued and not resuming until there is a response, frees up valuable milliseconds on the IMS region that can be in turn used to process more transactions.

To this end we will make use of the final two reengineering solutions (sections 2.4 and 2.5). Their joint purpose is to reestablish a call stack in the epilogue that corresponds to where the prologue left off. The selling point of this solution is simplicity, therefore it uses the cheapest variants of all the proposed solutions. These are storing the full state in memory and having the control flow refactored automatically.

Table 3.3 Simple asynch. callout solution evaluation; Legend: ● - desirable, □ - satisfactory, × - undesirable
Table 3.3 captures the resulting effects. While the cost of the solution has not changed dramatically, resource blocking has been reduced. Clearly, all this can only be achieved at the expense of other benefits. It is recommended for each project to judge whether the newly introduced drawbacks are acceptable. However, under normal circumstances, the drawbacks are very unlikely to act as show-stoppers.

### 3.4 Asynchronous Callout

As described in sections 2.4 and 2.5, there are many ways to separate the processes. The previous approach relied on the most cost-effective ideas, in contrast this one is made up of the most complete solutions. Firstly, the state is reduced to contain only dirty data, thereby optimizing the retained data volume. Secondly, the state is persisted during the remote call in order to recover from system crashes. Finally, manual code separation serves to preserve or improve code quality.

Note that code separation is the only exception to the continuity of the solution spectrum. Unlike other components that build upon previous solutions, this one has to be implemented not as a complement to but rather instead of code continuity.

<table>
<thead>
<tr>
<th>Distributed updates</th>
<th>Effort</th>
<th>Knowledge</th>
<th>Maintenance</th>
<th>Efficiency</th>
<th>Locking pitfalls</th>
<th>Data volume</th>
<th>Recovery</th>
<th>Code quality</th>
<th>Resource blocking</th>
<th>Chain lock reactions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>□</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Object immutability</td>
<td>□</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Local updates</td>
<td>□</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Persist dirty data</td>
<td>□</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Code separation</td>
<td>□</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Asynchronous callout</td>
<td>□</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 3.4 Asynchronous callout solution evaluation**; Legend: - desirable, □ - satisfactory, × - undesirable

Table 3.4 highlights the dramatic combined impact of the fully asynchronous approach on all cost factors. Due to these effects, it is recommended to carefully evaluate the necessity of each individual improvement over its simpler counterpart. Since the components are independent of each other, every one of them can and should be applied on a strict need-to-have basis.
3.5 Summary

The problem of assembling solutions to individual challenges has been tackled by introducing a cost-dependent spectrum of approaches ranging from a synchronous callout to a fully asynchronous option. Each step primarily aims to enhance the essential factors, namely the threat of locking out other applications and the tendency to block computing resources. The subsequent goal is to optimize other criteria, for example code quality.

Because each step is more accomplished in the two directions, it is naturally also more expensive than the earlier step. Importantly, each additional step builds upon the previous solutions, which means that there is always a possibility to upgrade to the next option without having to give up any progress. This also means that the spectrum (see Figure 3.1) acts as a roadmap towards the chosen stage. The intermediate steps can then be viewed as testable checkpoints and even possible resorts in the event of excessive financial or time pressures.

In the direction towards the fully asynchronous callout, solutions are more loosely coupled and provide higher throughput while sacrificing some performance, i.e. the runtime of each logical transaction.

All evaluation aspects of the solutions on the spectrum are symbolically summarized in Table 3.5. This comprehensive overview is intended as a guiding reference for project managers seeking to choose the most appropriate solution.

<table>
<thead>
<tr>
<th></th>
<th>Cost</th>
<th>Other criteria</th>
<th>Essential</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Effort</td>
<td>Knowledge</td>
<td>Maintenance</td>
</tr>
<tr>
<td>synchronous callout</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>split transactions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>simple asynch. callout</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>asynchronous callout</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.5 Full solution spectrum evaluation summary; Legend: ● - desirable, □ - satisfactory, × - undesirable
4 Reengineering Patterns

The main contribution of this thesis is capturing the reengineering process towards an asynchronous callout in terms of patterns. Patterns as a means to standardizing common processes in the field of computer science were first introduced by E. Gamma et al. in [GE95]. The idea of a pattern is to communicate not only the techniques but also the motivation and rationale behind them. Reengineering patterns in particular document the process of discovery and transformation in legacy code [DS08].

In order to conform to the quality criteria listed in [DS08], the focus of the patterns introduced in this section is twofold. Firstly, it is to clearly expose the advantages, cost and consequences of the target solution. The second objective is describing the process, i.e. how to get from one state of the system to another. Since the patterns are new, which means they have not had time to mature, an additional emphasis is placed on mutual independence. The expected benefit is being able to change or replace a pattern altogether without compromising the full solution.

The reengineering patterns below follow the format introduced in [DS08] on a sample pattern which is included for reference in Appendix 9.1. In addition to the listed descriptors, the patterns also contain estimates of the amount of work in person/days. These estimates were created from personal experience with respect to the Proof of Concept (PoC) implementation. Note that the transaction considered in the PoC is complex and highly entangled and may therefore be considered a good representative of applications in Credit Suisse. Furthermore, the patterns explicitly address the corresponding maintenance requirements as these make up one of the cost factors when it comes to reengineering (see section 2).

Content-wise, this section formalizes fully descriptive technical patterns. These patterns guide the transformation process required to achieve each high-level solution from chapter 2 that is necessary for the Simple Asynchronous Callout. A complete picture capturing the relations between the concepts and patterns is provided in Figure 4.1. The patterns already incorporate lessons learned in the PoC implementation (see section 5). When creating the patterns, the focus was to make them well-defined and mechanical. On one hand this means the programmers can execute them routinely. However, it also means that in some situations the patterns may produce suboptimal results. Experienced developers can consider optimizing the transformations on a case-by-case basis. Some optimizations were already performed in the Proof of Concept and are explained in section 5.
Figure 4.1 Pattern hierarchy. Vertically stacked patterns within a box are alternative solutions. Gray indicates that some patterns required for the concept are not available yet.
4.1 Launch Remote Notifications

**Intent:** Create externally callable notification procedures in the programs that perform the callouts. Launch these from the main program in its last callback.

**Estimates:** 1 person/day per notification

**Problem**

In order to preserve data consistency, remote services may implement either the reservation or the compensation pattern. From the local perspective the first requires that a confirmation message is sent if the logical local transaction succeeds whereas the latter requires an undo notification in case of an application or technical error.

**Solution**

The two notification types can be engineered similarly. From the point of view of the local transaction, sending these notifications boils down to calling a managed interface which underneath performs a fire-and-forget asynchronous call. Since the service behind this interface is closely related to the callout service, it is sensible to access it in the same module. Therefore, the core of this pattern is to create a new procedure in the callout module such that the procedure will have an external interface and will perform the notification call. To make this solution work, it is also necessary to pass the notification data to the main program via the intermediate modules and finally engineer the main program to invoke the new procedure. A sample dependency scheme of the end-product is available in Figure 4.2 below.

**Figure 4.2 Module dependencies after applying the pattern.** CCCCCC and BBBBBBB are intermediate modules on the callout path.

The first task is to create the new procedure. Its sole focus is to invoke the notification interface. A very small example of such procedure is shown in Listing 4.1. All data required for this purpose has to be passed via a based structure (line F), the pointer to which is supplied as an argument to the procedure (line D). The data can be of two kinds: first is the application data that is prepared in the normal flow of the same module, e.g. the correlation number that indicates which request is being confirmed/undone (lines C,G). The second type are IMS session parameters, for
instance the address of the queue manager control block (line H). The latter type of parameters will be supplied by the main program because it depends on the session and may differ in the different local physical transactions.

```
AAAAAA: proc(P_A_DATA);
    dcl A_DATA based(P_A_DATA) type A_STRUCT;
    dcl X_CALLOUT_DATA type X_CALLOUT_STRUCT;
    dcl NOTIF_DATA based(A_DATA.P_NOTIF_DATA) type NOTIF_STRUCT;/*A*/

    call XXXXXX_CALLOUT(ADDR(X_CALLOUT_DATA)); /*B*/
    NOTIF_DATA.CORREL_ID = X_CALLOUT_DATA.CORREL_ID; /*C*/
end AAAAAA_Normal;

AAAAAA_NOTIF: proc(P_NOTIF_DATA); /*D*/
    dcl NOTIF_DATA based(P_NOTIF_DATA) type NOTIF_STRUCT; /*E*/
    dcl X_NOTIFY_DATA type X_NOTIFY_STRUCT; /*F*/

    X_NOTIFY_DATA.CORREL_ID = NOTIF_DATA.CORREL_ID; /*G*/
    X_NOTIFY_DATA.MQ_MANAGER = NOTIF_DATA.MQ_MANAGER; /*H*/
    call XXXXXX_NOTIFY(ADDR(X_NOTIFY_DATA)); /*I*/
end AAAAAA_Notification;
```

**Listing 4.1 Sample changes to the callout program.** Newly added lines are blue. `XXXXXX_CALLOUT` and `XXXXXX_NOTIFY` are interfaces of the remote service.

The original procedure in this module has to supply the application data. Hence, it has to be modified to use the same structure (line A) and fill the appropriate variables (line C). For efficiency and readability considerations, the use of pointers should be minimised when creating the notification data structure.

**Interface changes:** The notification structure will be passed from the main module to the callout module via a pointer. The reason is to isolate the intermediate programs from having to know the exact target structure. Therefore, the interfaces of all modules on the callout paths have to be extended with this pointer. Additionally, each intermediate program has to pass the pointer further down the callout path.

**Changes to the main program:** The role of the main program is to first allocate the notification structure (`NOTIFY_STRUCT` in the running example) and pass its address to the first intermediate module on the callout path - module CCCCCC. Moreover, it has to assure that the notification structure will be retained during the callouts.
Finally, it has to fill the required **IMS session parameters** and call the notification procedure `AAAAAA_NOTIF`.

The described changes have to be made for each reservation/compensation pattern. What differs between the two cases is when the final step is performed. Namely, in the confirmation pattern, notifications are sent before the last commit if the logical transaction was successful. The reservation pattern requires a notification similarly before the last commit/rollback but in case there were errors in the application.

**Maintenance**

Modifications described in this pattern have to be performed every time the need for a new notification arises. Other changes that may require maintenance include needing access to fewer or more variables in the notification procedure. In this case, corresponding changes will have to be made in the normal entry of the `callout` module or in the main program depending on whether the changes concern application or **IMS session parameters** respectively.

**Tradeoffs**

**Pros:** The pattern is easy to execute and is expected to require little maintenance. Thanks to keeping all communication with the remote service in one module, the resulting code structure is understandable.

**Cons:** The pattern couples the main module with the `callout` module and as a result, new requirements in the `callout` module may solicit changes in the main program.

**Dependencies and Related Patterns**

The solution in pattern 4.5 `Launch Late Updates from the Main Module` relies on similar structural changes; parts of the infrastructure may be reused there.

### 4.2 Derive a Maximum Set of Lock Candidates

**Intent:** Find a maximum set of candidates by considering objects which are write-locked upon read access and also ones that have been deferred to be updated in the callback.

**Estimates:** 2 person/days per application

**Problem**
The goal is to identify business objects (BOs) which have to be locked in-between the physical transactions in order to guarantee data consistency.

Solution

Most IMS database updates follow a simple scheme consisting of the GHU (get hold unique) call which acquires a lock on the resource, followed by a set of changes and finalized with a REPL (replace) call that makes the update. The two are typically called in one procedure and there is a relatively small chance that the callout will appear in-between the two database commands. To find this type of lock candidates, it is necessary to examine the code surrounding the callout path to see if any of the procedures makes a GHU call prior to the callout and a REPL call afterwards. If this is the case, an application lock is necessary for the resource.

There are, however, more complex objects which obtain a database lock as soon as they are read. This type of privilege is defined in the PCB (program control block) of the main program using the PROCOPT=A option. An object from a database for which the program has this access privilege has to be locked on the application level as well. In order to find out which modules in the dependency hierarchy read and modify this database, the fastest way is to ask the application owner.

The final set of application lock candidates comes from the pool of postponed updates. Similarly to the previous category, these objects were already read in the prologue. It is likely that the result of the reads has played a role in the application code, which means the values should remain intact. However, if it is clear at first sight that the read results are not used in the code other than for the purpose of changing the object and performing an update, the objects can be disregarded. An example of such processing type could be a program setting its state in a database for monitoring purposes. Here the program itself is not concerned with what the previous state was and only cares about bringing it up to date.

For all the described candidates it is further necessary to find out whether one should expect concurrent access to the objects. Once again, the most effective source of this knowledge will be the application owner.

Output

The findings acquired in this reverse engineering pattern should be captured in a table similar to Table 4.1 below.
Table 4.1 Template for the summary of lock candidates table with explanations

**Maintenance**

While not strictly necessary, it might be desired to keep the table up to date, which means capturing newly introduced lock candidates.

**Tradeoffs**

*Pros*: The pattern is straightforward and efficient.

*Cons*: It relies on the availability of experts; the resulting set of candidates may be largely overestimated.

**Dependencies and Related Patterns**

The output of this pattern is used to introduce application locks, for instance by pattern 4.3 "Introduce Modify Tickets".

### 4.3 Introduce Modify Tickets

**Intent**: Reengineer the database reading module to return a hash of a business object upon read and compare this with the hash of a new read prior to an update.

**Estimates**: 3 person/days per lock candidate

**Problem**

Follow Credit Suisse standard ST_0428-014: Ensure isolation through modify tickets:

> "In public services a distributed transaction is not allowed. To achieve a non-blocking isolation, modify tickets are used. A read service can return a modify ticket. Modifying services accept the modify ticket as input. Consistency is ensured by confirming the ticket received with the state of the data." [TK08]

**Difficulties**: Re-reading the data is difficult because there are several layers of indirection between the reading module and the main module. These intermediate modules have functionality that can not be assumed to be idempotent, which means it can not be simply repeated. At the same time, the updating call path does generally not posses the functionality to replicate a read.
Solution

This pattern presents an approach that requires very little knowledge about the system. Much like a few other patterns in this collection, it produces dependencies as shown in Figure 4.3 below. The key idea is to extract the arguments passed to the reading/updating module and reuse them to perform a fresh read in the callback. However, some of these arguments might have to be slightly modified. An understanding of the argument list is hence the only specific knowledge required for a successful execution of the reengineering process.

![Figure 4.3 A sample resulting dependency structure.](image)

The first task is to add a hashing functionality to module TTTTTT which is responsible for reading/updating a business object (BO). This means that when the object is read, it will be possible to request its hash value. Here, the only real task is to call a standard hash module with the data of the business object. Additionally, the module’s interface may be extended with a hash request bit and the return value.

**Interface changes:** The next step is passing the arguments for module TTTTTT back to the main program DDDDDD. This will be done by allocating the communication structure in the main module and passing its address down the hierarchy of intermediate modules towards the last one, which in our example is SSSSSS. What this means is adding a pointer to the interface of each intermediate module and passing it as an argument to the next module. Finally, the allocation of the communication structure for TTTTTT in module SSSSSS must be changed to a based allocation based on the supplied pointer. These changes will make the communication structure available for reuse. Additionally, the interfaces have to return the hash value back to the main module.

**Main module:** The main module will play two roles in the modify ticket system. First, it has to allocate the communication structure as discussed above. Second, it has to change some arguments and finally perform the comparison by calling TTTTTT again.
in the callback and comparing the hash values. Importantly, the PCB that is used for the comparison read should be set up in a way that performing a read creates a physical write lock on the object.

Let us analyze the required changes to the communication structure. The foremost task is to update all IMS session parameters. As a rule of thumb, these are typically the parameters supplied as pointers. An equally important modification is making sure that the fresh read does not overwrite the existing read. To this end it is necessary to change the arguments which are used to pass the resulting BO to a new set of variables. Note that these are typically also supplied as pointers, which means a fresh set has to be created for the new read. The remaining task is likewise related to memory allocations. In case the BO is allocated dynamically in TTTTTT, the allocation has to be freed to avoid memory leaks. If this is the case, it is likely that the arguments will contain a bit indicating whether the memory should be freed. Since we are only interested in the hash value, this bit has to be flipped to true.

**Maintenance**

In case the interface of the reading/updating module changes, this may entail corresponding changes to the main program. At the very least, a review of the changes has to be performed to see if they affect the solution.

**Applicability**

This pattern produces an error if the comparison does not succeed. Hence, it is not suitable for BOs which are expected to be under concurrent write access.

**Tradeoffs**

**Pros:** The pattern is simple and relies on a bare minimum of specific knowledge.

**Cons:** It produces a tight coupling between the reading and main modules as shown in Figure 4.3. The new dependencies in turn create a stronger need for maintenance.

**Alternative solutions**

An ideal solution would be to extract the code which creates the argument list for TTTTTT and be able to invoke it (not duplicate it!) in the updating chain. This way, there would be no overarching dependencies as the comparison could be handled by a closer layer. Such refactoring would require a detailed knowledge of the code.
For **IMS** databases, the read has to be repeated in the callback anyway in order to acquire a write privilege. The hash value could simply be compared from this read.

**Dependencies and Related Patterns**

Pattern 4.2 "Derive a Maximum Set of Lock Candidates" can be used to see which **BOs** have to be locked. The resulting dependency structure of this pattern is similar to that of 4.1 "Launch Remote Notifications" and 4.5 "Launch Late Updates from the Main Module".

### 4.4 Breadth-First Search for Relevant Updates

**Intent:** Examine the code hierarchy for reachable updates manually layer by layer.

**Estimates:** 10 person/days per application

**Problem**

The location and nature of updating operations that occur before the **callout** are pivotal pieces of information for the asynchronous solution. It is necessary to discover these details about all forms of updates that can occur in background **IMS** transactions, namely database updates and queue puts.

**Difficulties:** Updates are likely distributed throughout a large hierarchy of modules. However, many of the updates in these modules are in fact not reachable via any call paths from the main program. Finally, some are irrelevant for the application logic.

**Solution**

The knowledge is built up gradually through examining the modules on each layer of the dependency graph before proceeding to the next one. The first layer is the main module itself. A comprehensive list of its dependencies, i.e. modules that are called directly from it can be obtained using the cross-dependency tool **TSO RIX** with a top-down query method and level 1. The resulting hierarchy should be represented graphically as exemplified by the sample scheme in Figure 4.4.

The goal is to filter out such modules on each layer that are called only after or instead of the **callout** or are irrelevant for the application logic. For each dependency, the latter criterion is the easiest to evaluate. An experienced **PL/I** expert can easily recognize whether a module is of technical nature. These types of modules perform
what in the technical jargon at Credit Suisse is referred to as "tools and puts". They can be eliminated from the model immediately.

---

**Figure 4.4 Sample full 2-layer dependency hierarchy**: can be derived from the output of the cross-dependency tool. The currently considered layer is highlighted.

If a module is deemed relevant for the application logic, the next step is to map the context in which it is used in the previous layer. This is done by recursively tracing the flow to its calling procedures until all paths from the main function are found. These are then matched with the known paths that are reachable prior to the callout.

Certain modules have many different modes of interaction. To illustrate, many databases are accessed through a common application-level module which may serve to retrieve a BO as well as update or delete it. However, each call can only do one of these operations. Clearly, these modes entail different call paths and more importantly different reachable DB operations. The existence of these modes can be revealed through a careful examination of the interface of the module. For each module it is necessary to note down the currently known set of reachable modes.

---

**Figure 4.5 A sample stripped 2-layer hierarchy**. Irrelevant or unreachable modules have been removed and reachable interaction modes are noted.

In each module we are ultimately looking for occurrence of the following statements:

- **PLITDLI**: typically used to perform a database call
- **EXEC SQL**: used to communicate with DB2 databases
- **MQ* (MQCONNECT, MQOPEN, MQPUT, etc.)**: communicates with queues

**Output**
A complete reachability graph including database and queue resources along with the type of access to these resources (read/write).

**Maintenance**

This being a reverse engineering pattern, the deliverable does not necessarily have to be maintained. However, once such a graph has been created, it is valuable and should be kept in accord with modifications to the system.

**Tradeoffs**

**Pros:** The approach is highly systematic and independent. The resulting graph is comprehensive.

**Cons:** Even if every layer is trimmed, the process gets exponentially harder and is therefore potentially inefficient. Additionally, an experienced PL/I expert is needed to tell technical modules from the ones that contain application logic.

**Improvements**

The application owner typically has a very good overview of these dependencies and can be of much use in any or all parts of the task. Involving the owner would significantly speed up the process and make it more robust to accidental errors.

A code analysis tool that produces static call graphs for PL/I would also be a very good alternative. For future reference, it might be worth looking into IT Panoramio.

**Dependencies and Related Patterns**

The outcome of this pattern is valuable in many different ways. Among others, it is directly relevant for update handling patterns such as 4.5 "Launch Late Updates from the Main Module" and 4.2 "Derive a Maximum Set of Lock Candidates".

### 4.5 Launch Late Updates from the Main Module

**Intent:** Split off updating functionality into separate entries and invoke all deferred updates from the main program.

**Estimates:** 2 person/days per update

**Problem**
All relevant updating interactions with databases and queues have to be postponed to the last executed physical transaction.

**Difficulties:** The existing updates are distributed in many different often shared modules far down the call hierarchy. Moreover, within these modules they might exist in a complex context of code statements.

**Solution**

For each update, firstly isolate the minimal updating code into a separate procedure with an external entry. Then augment module interfaces along the call path towards the update location with a bit indicating whether "to update or not to update" and a set of return parameters that are necessary to execute the update later. Finally, the main module has to be engineered to call the created external updating procedures prior to committing the last part of the logical transaction.

A typical update consists of the read and the update calls in close proximity. If the database is complex, these calls may be executed through the database's driver module instead of directly using the IMS PLITDLI or DB2 EXEC SQL statements. For the purposes of this pattern, the existence of such a layer of indirection is irrelevant. The two database calls and all code in-between has to be extracted into a separate procedure. Should they be immediately preceded by declarations that are in obvious exclusive relation to them, for instance a database query structure declaration, then these may also be placed into the new procedure.

The argument of the new procedure will be one pointer. The procedure is a level-1 procedure with an external interface, which means it cannot use variables declared in other procedures. Hence, all such required variables have to be declared globally if possible; otherwise they can be supplied as additional optional arguments. In any case, the target of the pointer argument will be a structure that holds all data necessary to complete the update operation later on.

The new procedure will be used in three modes: "no updates", "only updates", "everything". The first and second will be executed as part of an asynchronous solution and are distinguished by a mode bit in the pointer target structure, whereas the latter mode will exist for backward compatibility with other regularly structured modules and will come to effect in case the supplied pointer is null. Once the mode is known, only the relevant parts of the procedure can be executed. To this end, it is necessary to analyze the code for read-relevant vs. write-relevant statements. For
instance, statement B in Listing 4.2 below has no relevance for "only updates", whereas in contrast statements C through F can be omitted for "no updates".

```
UPDATE_STUFF: proc(POI);
   dcl QUERY type STUFF_QUERY init(/* ... */);
   dcl RESULT type STUFF_RESULT;

   call PLITDLI(a,'GHU ',DB_PTR,ADDR(RESULT),QUERY); /* A */
   RESULT_TYPE = RESULT.TYPE; /* B */
   if (RESULT.VALUE < DESIRED_VALUE) then do; /* C */
      RESULT.VALUE += 42; /* D */
      call PLITDLI(b,'REPL',DB_PTR,ADDR(RESULT)); /* E */
   end; /* F */
end UPDATE_STUFF;
```

**Listing 4.2 Simplified sample update procedure:** blue colour highlights globally declared variables.

The classification of statements gives rise to a corresponding relevance of global variables. The RESULT_TYPE variable is write-only, the DESIRED_VALUE is only used when updates are requested and finally DB_PTR is used in all modes. The values of update-relevant variables have to be made available in the "only update" mode. To this end, a copy will be stored in the POI argument's target structure shown below.

```
dcl 1 UPDATE_STRUCT based(POI),/* if POI is null->"everything" */
2 UPDATE_MODE     bit,       /* 0="no update", 1="only update" */
2 S_DESIRED_VALUE bin(32),   /* S stands for save */
2 DB_PTR          pointer;   /* IMS session dependent! */
```

**Listing 4.3 Sample update retention structure**

Retention structures like the one in Listing 4.3 serve as a communication basis between the main module and the update procedures. They are declared by the main module with the UPDATE_MODE initially set to 0 and the pointer declared in the example as POI is passed through the entire call hierarchy. To engineer the pointer passing mechanism, it is necessary to add the pointers to the interfaces of the updating structures and then recursively to all calling modules until the main module is reached. The advantage of passing pointers rather than full structures is that modules on the call path are shielded from having to know the corresponding types.

The remaining task is to engineer the main module to perform all update calls. Firstly, this means all update retention structures have to be declared in the prologue.
with UPDATE_MODE set to 0. Then, it is necessary to distribute their addresses into the communication structures on the call paths. Finally, a new subroutine in the main module has to be created which will perform the actual calls. Additionally, it has to fill IMS session parameters such as DB_PTR where necessary. This subroutine has to be performed in the final part of the logical transaction.

**Maintenance**

Maintaining this pattern is a challenging task. What it requires is that all newly introduced updates are reengineered in the described manner. Therefore the major problem is in fact making developers aware that by introducing updates in shared modules there is a chance these updates might be relevant for the asynchronous solutions and therefore have to be engineered to fit the asynchronous model.

A more subtle maintenance need arises from the tight coupling that has been introduced between the updating module and the main program. It is the need to supply IMS session parameters to some update retention structures. This means that in the unlikely event that the updating procedure is altered to require new IMS session parameters, the change has to be reflected in the main module. Fortunately in this case, the developers can be made aware of this requirement by putting comments in the copybooks that declare the structures. The actual engineering task is then relatively straightforward.

**Tradeoffs**

**Pros:** The approach preserves the physical location of the updating code and does not introduce any code duplications. Because the final update calls circumvent the intermediate modules, these need only minor changes.

**Cons:** The pattern couples the main module with update operations which can change easily, thus introducing maintenance problems.

**Dependencies and Related Patterns**

The pattern relies on a full knowledge of relevant updates. 4.4 "Breadth-First Search for Relevant Updates" describes a possible solution for this requirement.

**4.6 Base Variables in Storage Areas**

**Intent:** Prepare grounds for state retention by allocating all state variables in per message memory areas.
Estimates: 4 person/days per application

Problem

Regardless of the method in which the state is eventually stored, there has to be a mechanism to consolidate the state-relevant variables. The state is expected to incur frequent changes, so the solution has to be maintainable with ease.

Solution

Aggregate variables by employing the PL/I AREA construct which allows storing data in a specific section of memory. Areas can be copied into different parts of memory and saved into files atomically, which makes them a good basis for state storage.

The technical approach is to store the state variables of all modules on the callout path in one self-defining area. This means that the first few bytes of the area indicate the relative locations of all variables allocated in this area. Hence, recovering the structure of allocations becomes simple and independent of the IMS region.

This pattern assumes the existence of a storage module with the following interface.

```plaintext
STORAGE: procedure(ACTION, AREA_PTR, TOKEN, IMS_PTR, FAILURE_CODE);
/* ACTION: 'I' = Init, 'S' = Save, 'R' = Restore, 'F' = Free */
/* AREA_PTR: pointer to the memory area for the current LUW */
/* TOKEN: a token identifying the area */
/* IMS_PTR: pointer to a structure of the IMS session pointers */
/* FAILURE_CODE: indicates success or failure of the operation */
```

Listing 4.4 Interface of an assumed storage module; IMS pointers are supplied for the sake of handling states that contain pointers, e.g. to avoid freeing session pointers.

The meanings of the different storage actions are detailed in Table 4.2 below.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>Return a large enough empty area and an identifying token.</td>
</tr>
<tr>
<td>SAVE</td>
<td>Save the area including recursive pointer destinations.</td>
</tr>
<tr>
<td>RESTORE</td>
<td>Restore the area with the same structure and valid pointers.</td>
</tr>
<tr>
<td>FREE</td>
<td>Free the recursive pointer targets and the area itself.</td>
</tr>
</tbody>
</table>

Table 4.2 Storage module actions

The reengineering steps in this pattern are designed to first initialize a template area structure, in which variables from the main program are allocated. The template area is then atomically copied to the initialized working area in every message loop
iteration. Finally, variables in all modules on the callout path are also based in the area and conditionally allocated in the prologue.

**Declarations:** Declaration of the template area in the main program has to be done in the beginning – prior to declarations of state variables. Listing 4.5 below shows an allocation process of the template area and an offset structure as the first allocation inside of the area.

```plaintext
dcl P_DATA_AREA ptr;
dcl DATA_AREA AREA based(P_DATA_AREA);

dcl O_DATA_AREA_OFFSETS offset(DATA_AREA);
dcl 1 DATA_AREA_OFFSETS based(O_DATA_AREA_OFFSETS),
   %INCLUDE OFFSETSTR; /* include the offset structure */
allocate DATA_AREA; /* allocate the initial data area */
allocate DATA_AREA_OFFSETS; /* allocate the offset structure */

dcl P_TEMPLATE_AREA ptr init(P_DATA_AREA); /* store the template */
```

**Listing 4.5 Template area declarations**

**Allocations:** To allocate all state variables in an area, they have to be converted to based variables. Each variable will be based on an offset that is part of the offset structure like the one shown in Listing 4.6. Basing an automatic variable requires adding an offset with a corresponding name to the offset structure and augmenting the existing declaration with the "based" keyword (see Listing 4.7). The procedure for static variables is similar. Existing based variables do not need to be based again, it is sufficient to base their locator (which is typically automatically allocated). The final case, controlled variables, are very rare and have to be considered on a case-by-case basis.

```plaintext
6 O_VAR_A           OFFSET(DATA_AREA), /* char(100) */
6 O_VAR_B           OFFSET(DATA_AREA), /* STRUCT_1 */
```

**Listing 4.6 Excerpt from a sample offset structure.** All offsets are relative to the DATA_AREA, each offset comes with a comment about the target type.

Note that offsets in the offset structure are declared in the DATA_AREA. Hence, changes to the P_DATA_AREA pointer conveniently make all variables refer to their allocations in the new target of P_DATA_AREA.
Listing 4.7 Converting an automatic to a based allocation. For variable inside the message loop, allocations have to be conditionally performed only in the prologue.

Note that offsets in the offset structure are declared in the DATA_AREA. Hence, changes to the P_DATA_AREA pointer conveniently make all variables refer to their allocations in the new target of P_DATA_AREA.

Storage invocation: The main program coordinates data storage by invoking the storage functions in every message loop. The "init" function is called at the beginning, passing P_DATA_AREA as the area pointer. The newly allocated area is then assigned the contents of the template area (identified by P_TEMPLATE_AREA). If a callout is performed, the save method has to be invoked prior to terminating the loop. The token has to be passed in the request and response messages in order to allow for retrieval of the area using the "restore" command. Here once again, the P_DATA_AREA is passed to the storage module. Finally, regardless of whether the callout was performed, the state has to be freed when it is no longer needed.

Module interface changes: Since all modules on the callout path need to allocate their variables in the storage area, they need to gain access to it via arguments. Therefore, the interface of each module along the path is extended with an area pointer. Importantly, the fact that all modules use the same area requires a careful structuring of the offset pointers. It is good practice to ensure that a called module does not need to know anything about its caller. Therefore, the offset pointers will be structured from the offsets of the last module on the callout path towards the main module. This way, each module along the path can overlay the offset structure exclusively with offsets for modules that are further in the callout path.

Listing 4.8 Organization of offset structures; Module BBBB would overlay the memory with offsets for AAAAAA and itself only, etc. Module names correspond to Figure 4.2.
In order to preserve the intended separation of modules, it is forbidden to use the storage area as a means of communication between modules. All data has to be passed through arguments.

The solution needs to be further augmented with a pointer target saving mechanism.

**Maintenance**

If a new state-relevant variable is introduced, it is required to be based in the offset structure and fit into the provided working area size.

**Tradeoffs**

**Pros:** The pattern is relatively easy to execute and provides a good state aggregation mechanism and a clear separation from the storage module.

**Cons:** Explicit allocations have an adverse impact on code readability. Moreover, the use of dynamic allocations introduces a runtime overhead.

**Dependencies and Related Patterns**

The pattern assumes the existence of a storage module; a possible candidate is pattern 4.7 "Save State". Pattern 4.8 "Save Pointer Targets of Known Types" offers a possible add-on for pointer saving.

### 4.7 Save State using the Token Service

**Intent:** Retain state in shared memory using the token service.

**Estimates:** 1 person/day per application

**Problem**

After the remote response arrives, it can be scheduled to be processed in any IMS region. Moreover, the program is only allowed to process a limited number of messages in one IMS region before being cleaned up by the scheduler. Hence, it is necessary to preserve the state more durably, which means beyond the address space of a single IMS region.

The storage module has to implement the interface from Listing 4.4 and Table 4.2.

**Solution**
Store the state in areas in shared memory space managed by the token service. The storage module implements the specified interface by communicating with the assembler interface of the token service. This module can be standard and reusable.

**Maintenance**

The solution does not have any logic that is specific to the project. Therefore apart from keeping up with requirements for area sizes, there is no need for maintenance.

**Tradeoffs**

*Pros*: The solution is efficient because all allocations are performed in the storage area from the start, so there is little copying and manipulation required.

*Cons*: As opposed the approach of persisting the state in a more reliable storage, there is an increased risk of losing the data and causing all ongoing logical transactions to have to be manually restarted.

**Dependencies and Related Patterns**

The storage module expects variables to be stored in PL/I areas – for this one could use pattern 4.6 "Base Variables in Storage Areas".

### 4.8 Save Pointer Targets of Known Types

**Intent**: Store recursive pointer targets in a storage area by iterating over all known pointers in the state variables.

**Estimates**: 2 person/days per application

**Problem**

With high probability state variables include pointers to dynamically allocated storage. Considering that in such cases the allocations are made in possibly unexplored modules far down the call hierarchy, regulating the allocations would pose a substantial reengineering overhead. These dynamic allocations are made in the address space of the encompassing IMS region; therefore they have to be moved to shared storage along with the rest of the state.

**Solution**

Given that the types of all pointer targets and the types of pointers within the targets etc. are known, the solution in this pattern is to recursively allocate and copy the
target structures into the storage area while avoiding duplication of data in case the pointers are non-injective.

In order to achieve this goal, the first step is to find all pointers in the recursive state. The next step is to save the target structures in the pointer tree in a post-order manner. To illustrate, Listing 4.9 below shows a set of structures that form a pointer tree. Assuming that the based variables have been allocated, the post-order algorithm would visit the targets in the order: D, B, C. A is based on a storage offset, which means it has already been allocated before.

Listing 4.9 Sample pointer tree; code is displayed in 3 separate columns.

dcl 1 A based(O_A),
2 PTR1 pointer,
2 PTR2 pointer,
2 MOREDATA char(2);
dcl 1 B based(PTR1),
2 PTR3 pointer,
2 BDATA bin(31);
dcl 1 C based(PTR2),
2 CDATA char(7);
dcl 1 D based(PTR3),
2 DDATA dec(2);

dcl 1 A based(O_A),
2 PTR1 pointer,
2 PTR2 pointer,
2 MOREDATA char(2);
dcl 1 B based(PTR1),
2 PTR3 pointer,
2 BDATA bin(31);
dcl 1 C based(PTR2),
2 CDATA char(7);
dcl 1 D based(PTR3),
2 DDATA dec(2);

Listing 4.10 demonstrates what it means to visit a target. The example changes the target of pointer POI which is a state variable directly based on an offset (line A). It is known that the target type of POI is POI_TYPE (line B). The ALREADY_USED function on line C checks whether the target has already been reallocated and if so, changes POI to point to the new target. Otherwise, a new allocation is made in DATA_AREA (line E), its contents is copied from the old one (F) and the old allocation is freed in line G. Finally, the old and new pointers are saved on line H to be available for future runs of the ALREADY_USED function.

Reallocations are done based on allocated sizes rather than the target structure sizes because it is possible that the original allocations were made for different types and are only being overlaid with the true target structures.

The full tree is therefore reconstructed in the storage area. Unfortunately, there is no way to iterate over structures in PL/I, therefore the post order of the tree has to be prepared manually. Returning to the example from Listing 4.10, the visits would have to be scheduled over the <pointer, target> pairs: <PTR3, D>, <PTR1, B> and <PTR2, C>. Once this is done, the full tree has been successfully reallocated.

When saving an area, a special attention has to be paid to IMS session parameters, which are session-wide and should therefore not be freed. Restoring an area in an IMS region is simple as the allocations can stay in the shared area. The only
overhead is in updating all state pointers which should be equal to **IMS session parameters** to the values of the valid pointers from the current session.

```plaintext

dcl POI pointer based(O_POI); /* A */
dcl POI_STR based(POI) type POI_TYPE; /* B */

if ALREADY_USED(POI) = '0'b then do; /* C */
    AUX_POI = POI; /* D */
    allocate allocsize(AUX_POI) in STORAGE_AREA set POI; /* E */
    POI->POI_STR = AUX_POI->POI_STR; /* F */
    plifree AUX_POI; /* G */
    ADD_USED(AUX_POI, POI); /* H */
end;

Listing 4.10 Sample pointer reallocation for one pointer target
```

**Maintenance**

Clearly, this way of handling the pointers requires an intimate up-to-date knowledge of the state pointer tree. Hence, upon every change to the pointer structure, post order scheduling has to be updated.

**Tradeoffs**

**Pros:** The reallocation algorithm is simple and complete. Additionally, the complexity is mostly in the saving operation, enabling an uncomplicated restoration process.

**Cons:** A tight coupling is introduced between the application and the structure of the pointer tree. Hence, the pattern is hard to maintain because developers who make changes to the structure may not be aware of this application’s requirements.

**Dependencies and Related Patterns**

The knowledge of the pointer tree can be gained for instance by following the pattern 4.9 "Breadth-First Search for Allocation Types".

**4.9 Breadth-First Search for Allocation Types**

**Intent:** Systematically examine modules on the dependency hierarchy for allocations.

**Estimates:** 10 person/days per application

**Problem**
Find out what type of structures are meant to be at the target of untyped pointers.

**Difficulties:** The targets could be allocated and referred to in any module out of an excessive number of the main program's hierarchy of dependencies. To further complicate matters, the modules that make these allocations may of course name the pointers differently and assign them in a complex flow of statements.

**Solution**

The module dependencies are examined in a breadth-first manner. On every level of the tree it is necessary to note down which pointers have been passed to each module and under what name they are known in the module. The search for each pointer's target type is complete when a location in the code is found where the pointer is assigned or directly allocated.

Table 4.3 below shows a sample of a tabular document that will be used to track the progress of the dependency graph crawl. Data is exchanged between modules only via arguments. Hence, to see which pointers are passed to a dependent module, it is necessary to discover all pointer arguments in this module's interface. Each of these should be supplied by the calling program and here is where the link between the names can be learned. To illustrate, program DDDDDD supplied its p_poi_str when calling program CCCCCC, which however declared this pointer under the name p_pointers. Once the type of a pointer target is known, the module responsible for its allocation is highlighted and the line is complete. In case this structure contains further pointers, a new line is created for each of them and the process continues.

<table>
<thead>
<tr>
<th>DDDDDD</th>
<th>CCCCCC</th>
<th>RRRRRR</th>
<th>LLLLLL</th>
<th>TTTTTT</th>
<th>...</th>
<th>final type</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_poi_str</td>
<td>p_pointers</td>
<td>p_pois</td>
<td>p_poi_str</td>
<td>-</td>
<td>...</td>
<td>POISTRTYPE</td>
</tr>
</tbody>
</table>
| * poi1 | poi1 | - | - | 45ATYPE | 4B_SEGTYPE | ...
| * poi2 | poi2 | - | - | ... | ? |
| p45a | - | - | poi2 | - | ... | 45ATYPE |

Table 4.3 Target types of the trading order pointers

We have learned how to explore the tree, the next task is being able to recognize allocations and their types. All allocations made in any module other than the main one are bound to be made dynamically in order to remain allocated beyond the end of the procedure. Hence, an occurrence of the keyword "allocate" or short "alloc" in a module indicates a potential allocation of a state pointer. What remains is to find the
type of the allocated variable and the local name of the pointer. Since the name should have previously been entered in the table, the identity of the pointer is clear.

**Output**

In addition to the completed table, the developer should also insert a comment next to each pointer declaration that contains the type of the target structure.

**Maintenance**

Should any pointer be added, the table and more importantly the comments have to be brought up to date.

**Improvements**

Involving an application owner in the process will make it more efficient as he or she is presumably familiar with the use of these pointers throughout the hierarchy.

The number of modules that have to be examined can also be reduced if it is known which paths are reachable before the callout.

A code analysis tool for PL/I would also be a very good alternative. For future reference, it might be worth looking into IT Panoramio.

**Tradeoffs**

**Pros**: The pattern is systematic and independent, it is also purely structural, which means it does not require any prior knowledge about the system. Because the progress is noted down step by step, the pattern can be executed cooperatively.

**Cons**: Because all dependencies are examined manually, the process is lengthy.

**Dependencies and Related Patterns**

The result of the pattern is necessary for pattern 4.8 "Save Pointer Targets of Known Types". The efficiency can be improved if only reachable paths are considered. These can be discovered using pattern 4.4 "Breadth-First Search for Relevant Updates".

**4.10 Discover Callout Paths Backwards**

**Intent**: Find out which paths in the code lead to the callout locations by examining the code backwards starting from the callout points.

**Estimates**: 1 person/day per application
Problem

In general, there can be a number of paths in the code that lead to the callout location. Being able to mimic the original control flow in the callback requires that all these paths are known.

Solution

The succession of modules that lead to the callouts can be determined using the cross-referencing tool invoked by the command TSO RIX. The purpose is to find module dependencies from the callout module backwards. Hence, the bottom up query method should be used. Listing 4.11 below shows a sample output of this tool invoked on the module AAAAAA.

```
[**] *AAAAAA     <TYP=PLU>
[1 ] *AAAAAA   BBBBBB     <TYP=PLU>
[2 ] *AAAAAA   BBBBBB   XXXXXX   <TYP=PLI>
[2 ] *AAAAAA   BBBBBB   YYYYYY   <TYP=PLU>
[3 ] *AAAAAA   BBBBBB   YYYYYY   ZZZZZZ   <TYP=PLO>
[2 ] *AAAAAA   BBBBBB   CCCCCC   <TYP=PLU>
[3 ] *AAAAAA   BBBBBB   CCCCCC   DDDDDD   <TYP=PLU>
```

Listing 4.11 Sample RIX tool output

This output corresponds to the dependency graph in Figure 4.6 below.

Now that the dependencies are known, the callout paths within each module can be examined individually. A suitable approach here is to recursively find calls to procedures that are already known to be on a path towards the callout. Consider the code in Listing 4.12, which highlights the callout paths in the code for CCCCCC, i.e. paths towards the next callout module BBBBBB. In order to find these paths, one will initially find the call to BBBBBB. Noticing that the surrounding procedure PROC_B is not the main procedure, the next step is to find all calls to PROC_B and so forth. This control flow contains 3 distinct callout paths which differ in the calls to PROC_B.
Listing 4.12 Sample control flow of a module

Output

The purpose of this reengineering pattern is to provide an overview of the callout paths. A simple way to capture the outcome is in an ASCII graph.

```
DDDDDD -> PROC_1 -> CCCCC

CCCCCC -> PROC_A -> PROC_B (3) -> BBBBBB
```

Table 4.4 Output of the Discover Callout Paths pattern. The full output should contain paths for each module on the callout path. Multiplicities of calls are included.

Maintenance

The output of this pattern is currently used for one-time purposes only. Therefore, there is no real need to maintain it. However, should such a need arise; the resulting table would have to be manually updated by any developer who makes relevant changes to the control flow. For this purpose it is recommended to maintain the graphs as comments in the header of each individual code file. This way, a physical disconnect between the file and its documentation is removed, resulting in higher chances of keeping the documentation up-to-date.

Improvements

IT Panoramio might provide a suitable alternative to manual code analysis.

Tradeoffs

Pros: The pattern is simple and fast to complete. It makes use of available tools to simplify the task at hand. It relies on and builds up a minimal necessary knowledge of the code base.
**Cons**: The maintenance may be cumbersome and the result with time progressively untrustworthy. However, since the pattern is quick to execute, a viable alternative to maintaining the output is to simply recreate it in case of need.

### 4.11 Line Callout Paths with Entry Points

**Intent**: Introduce callback entry points at each call statement along the callout paths. Wire callbacks to follow the callout path based on local decisions.

**Estimates**: 2 person/days per application

**Problem**

Before a callout is made, the control flow navigates via a certain path. The goal of this pattern is to retrace the same path in the callout so that the code can continue executing in the same way as if the callout did not interrupt the flow beforehand.

**Solution**

Notably, each callout path is made of local control decisions. In the example from Listing 4.12 even though there are 3 different paths, they only differ by the decision made in PROC_A about which PROC_B call is performed. Following this reasoning, the approach taken in this pattern is to reconstruct the full path by recovering the individual local decisions.

```plaintext
PROC_A: proc; /*... more instructions ...*/;
PROC_A_ENTRY = 1; /* A */
call PROC_B; /* B */
if CALLOUT_PERFORMED then do; /* C */
  return; /* D */
PROC_A_1: entry; /* E */
  call PROC_B_CALLBACK(PROC_B_ENTRY); /* F */
end; /*... more instructions (also more callbacks!) ...*/;
end PROC_A;
```

**Listing 4.13 Introducing entry points along callout paths**: the blue lines are added

In more technical terms, each procedure remembers the point at which the callout was performed. This point serves as the callback entry for the procedure. To complete the picture, the first instruction of a callback is to recursively invoke the callback on the next procedure on the callout path. The required changes are demonstrated in Listing 4.13.
Since the callback may be performed in a different IMS region, the physical address of the entry point may change. Hence the entry points are stored in an array and what is retained is the index of the appropriate entry. For the sake of readability, the indexing variable name and its value correspond to the callback name as demonstrated on lines A, E and F. PROC_B_CALLBACK is an array of callback entry points in PROC_B and it is indexed by the PROC_B_ENTRY variable. This pattern is repeated in all call statements along the callout paths. The new variables have to be declared in the beginning of the program as follows in Listing 4.14.

```plaintext
  dcl PROC_A_ENTRY dec(1); /* has to be part of the state! */
  dcl PROC_A_CALLBACK(3) entry init(PROC_A_1, PROC_A_2);
```

Listing 4.14 Sample callback helper declarations. A similar set of variables has to be declared for each procedure (except for the main one) on the callout path.

Note that the fact that a callout is reachable does not mean that it will be executed. Hence the global variable CALLOUT_PERFORMED on line C indicates whether the call indeed led to an asynchronous callout. The condition on line C allows the program to continue executing if the callout was not made.

Module interface changes: Additionally to internal changes in a module, some modifications to the communication between the modules on the callout path are necessary. Namely, inter-module communication is not based on entry points. Rather, it is typically a simple text-based indication of the interaction mode. Most often there is an 'I' mode for initialization and an ' ' mode for normal execution. In this pattern, another mode will be introduced: 'C' for callback. Moreover, an indication of whether the callout was performed is required at this point. A simple approach is to extend each module's interface to optionally accept the CALLOUT_PERFORMED variable by reference. See the form of the external call below.

Finally, the main procedure of each module on the callout path has to be refactored to understand the new interaction mode. While all variable declarations have to be performed both in the normal execution mode and in the callback, one may expect to find many logic-related computations in the main procedure that should only be executed once. As a rule of thumb, these are all statements performed after the variable declaration and the include block in the beginning of the main procedure. In fortunate cases, the procedure will include a comment indicating the start of the main processing line.
PROC_B: proc; /*... more instructions ...*/;
PROC_B_ENTRY = 1; /* A */
BCOMMSTR.MODE = ' '; /* normal execution */ /* B */
call BBBBBB(BCOMMSTR, CALLOUT_PERFORMED); /* C */
if CALLOUT_PERFORMED then do;
  return; /* D */
PROC_B_1: entry; /* E */
  BCOMMSTR.MODE = 'C' /* callback*/ /* F */
call BBBBBB(BCOMMSTR); /* G */
end; /*... more instructions ...*/;
end PROC_A;

Listing 4.15 Introducing entry points on external calls

If the amount of code in the main procedure is substantial, it is recommended to extract all logic-related statements into a separate procedure and equip it with entry points as described above. All that remains is to add a conditional statement which based on the mode calls either the full procedure or just the callback.

Maintenance

In order for the control flow to function properly, it is vital that the solution be kept up-to-date. This means ensuring that if newly added logic introduces new callout paths, these will get properly lined with entry points. Both of these requirements should be relatively easy to achieve using appropriately eye-catching comments to make the structure of the code explicit.

Tradeoffs

Pros: The pattern is quick and easy to implement. Because all decisions are localized to a small block of code, the pattern does not introduce any complex dependencies between the modules.

Cons: Though it only requires a few simple steps, maintenance has to be performed manually in order to keep up the coherence of the resulting code.

Dependencies and Related Patterns

This pattern assumes a complete knowledge of the existing callout paths. For a way to gain such overview, see e.g. pattern 4.10 "Discover Callout Paths Backwards".
5 Proof of Concept Implementation

The PoC implementation was done in cooperation with the XPP Brokerage project. It is one of the first projects at Credit Suisse to reengineer a local subroutine into a remote callout. At the time of writing, the project is in the first-release implementation phase. While the current objective is to create a synchronous callout as the duration of the remote roundtrip is not excessively long, the project leaders are aware that such solution may be insufficient in the long run. This is why they agreed to host the proof of concept. To my benefit, the project's PL/I developers had already built up a considerable knowledge about the system at hand. Additionally, their lead PL/I expert was available for frequent consults. XPP Brokerage created excellent conditions for familiarization with PL/I, as well as the Credit Suisse programming tools and methods and for the development of theoretical concepts and patterns. They also provided technical support for the actual development efforts.

The objective of XPP Brokerage is to offload a part of the functionality used mainly by the application WC0160 to a remote system. The program is executed as part of a large chain of processing named WS80-Börse which handles business orders. Within this chain, WC0160 coordinates the business-related computations pertaining to each order. One of these computations, namely pricing, will now be performed externally. Figure 5.1 shows the module dependency path of WC0160 that is most relevant for the purposes of this thesis because it calls the pricing engine.

The relevant flow of the program consists of reading the trading order from database P90 and passing this order through the callout chain to the pricing engine. The modified order is afterwards updated in P90.

5.1 Implementation

The scope and extent of the PoC implementation were adjusted to be manageable by a single Master student over the course of several weeks. Hence the goal was to
execute all applicable patterns possibly in a simplified form in order to transform the
system into a minimum run-able version of the simple asynchronous callout. Certain
patterns which are meant to be executed for all instances of a problem and are not
strictly necessary to run the application were done once for demonstration purposes.

Figure 5.2 shows a diagram of the PoC infrastructure that is simplified in comparison
with the proposed infrastructure in Figure 2.4. Incoming messages are stored in a
file. Copying them into the queue sets the transaction in motion. For simplification,
the callout is performed from the main program rather than the final module on the
callout path. This allows reusing the existing queuing infrastructure. The java
application simply forwards messages to the input queue. The messages are of
similar format as the normal incoming messages and contain a correlation token. All
PL/I modules have been given analogous names as the original modules. The callout
path was also shortened to 3 modules to reduce the reengineering workload.

Figure 5.2 PoC infrastructure

5.2 Reverse Engineering Results

Figure 5.3 shows the outcome of pattern 4.4 "Breadth-First Search for Relevant
Updates". The graph is trimmed after module YWBTAR because the XPP Brokerage
team had previously analyzed the layers below this program. Their results are
included for completeness as an appendix in Figure 9.3. What these graphs show is
that before the callout there are only 2 distinct database updating operations that
have relevance for the application logic. The first, an update on the P90 database
which holds the trading orders, is invoked from module YYWITST and is in fact
related to an existing application locking mechanism. In other words, it is used to
acquire an exclusive write access for a specific order that lasts across physical
transactions. The second update – performed directly in YWC0160 – is hence the only
true relevant postponement candidate.
Figure 5.3 Overview of relevant reachable updates from WC0160 before the callout; the exploration was trimmed at YWBTAR

As a side product, the analysis of reachable database updates also shows the reads. As discussed in pattern 4.2 "Derive a Maximum Set of Lock Candidates", all reads of database objects which implicitly acquire a lock are application lock candidates. An example from this category is the trading order from P90 which is read using a PCB declared with PROCOPT=AP (all parentage). The physical lock created in this pre-callout read that is invoked through YWBP90 has to be substituted with an application lock. Databases P40, P42 and P50 that are also read before the callout do not have the exclusive 'A' privilege. The other type of lock candidates come from the pool of postponed updates. Since the only one was found in database P2W called from module YWC0160, this is another lock candidate. Coincidentally, this database also has PROCOPT=A, which is another confirmation that the lock is needed.

<table>
<thead>
<tr>
<th>Database</th>
<th>Reading/Updating procedure</th>
<th>Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>P90</td>
<td>YYP90_HANDLE (YWBP90)</td>
<td>No</td>
</tr>
<tr>
<td>P2W</td>
<td>PLITDLI in EINF_AUFZAEHLEN (YWC0160)</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 5.1 Application lock candidates

Table 5.2 below shows the outcome of a partial demonstration of pattern 4.9 "Breadth-First Search for Allocation Types". It was executed with prior knowledge that pointer PWBWSA which belongs to the state of WC0160 points to a trading order.
and is therefore allocated somewhere in the dependency hierarchy of YWBTST. Interestingly, it turns out that the allocations are in fact conditioned on the type of record that comes out of the database. Hence, one pointer can refer to different record types. Fortunately though, none of these types contains any further pointers. This means that not the precise types but rather only sizes matter for the pointer target saving exercise.

<table>
<thead>
<tr>
<th>WC0160</th>
<th>YWC0160</th>
<th>YWBTST</th>
<th>YWBP90</th>
<th>Final type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWBPPOINT</td>
<td>PWBPPOINT</td>
<td>-</td>
<td>-</td>
<td>WBPOINT</td>
</tr>
<tr>
<td>PWBWSA(1)</td>
<td>PWBWSA(1)</td>
<td>PWBWSA(1)</td>
<td>WBBN90</td>
<td></td>
</tr>
<tr>
<td>PWBWSA(2)</td>
<td>PWBWSA(2)</td>
<td>PWBWSA(2)</td>
<td>WBDI90</td>
<td></td>
</tr>
<tr>
<td>PWBWSA(3)</td>
<td>PWBWSA(3)</td>
<td>PWBWSA(3)</td>
<td>WBST90/WBWS90</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>PWBWSA(12)</td>
<td>PWBWSA(12)</td>
<td>PWBWSA(12)</td>
<td>WBAN90(/02/03)</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2 An extract of the pointer target analysis

The final reverse engineering pattern is 4.10 "Discover Callout Paths Backwards". The already straightforward task was further simplified by the prior knowledge of the callout modules. The outcome is depicted in Listing 5.1 below. The calls from YWBTAR to the next layer of modules below YWBTAR had been gathered behind a common interface that is referred to here as the pricing engine. The analysis of callout paths shows that there are 7 paths and they only differ in which RECHNEN call was made from HAUARTSTEUER in module YWC0160.

WC0160 -> MSG_VERARBEITUNG -> YWC0160
YWCO160 -> HAUARTSTEUER -> RECHNEN (7) -> YWBRIF
YWBRIF -> YWBTAR
YWBTAR -> B02_CALCULATION -> pricing engine

Listing 5.1 Callout paths of WC0160

5.3 Forward Engineering Details

As the above analysis has shown, there is only one update postponement candidate. It is the P2W update in module YWC0161. The original code of the surrounding procedure is included as an appendix in Listing 9.1. The procedure was already exclusively focused on the update, so there was no need to extract any parts of it into a smaller procedure as suggested in pattern 4.5 "Launch Late Updates from the Main Module". A close look at the nature of the externally declared variables and arguments revealed that variables LAND, AART, TITEL, EINF_STOP_TXT and
VOLL_EINFZAHLER_BIT are used only used to update the external state. They have no impact on the updating code itself. In contrast, variable EINF_ZAEHLER is read-only and has impact on the update processing. Finally, variables CT4 and CT3 are constants declared in copybook WBIMSK, the based variable TPSTC is from copybook TPPCB and WBP2W is a session-wide pointer to the database. Additionally, copybook TPPCB requires an existence of a variable named LTM which is also a session pointer. Listing 5.2 shows the resulting callback input structure.

```define structure
1 WBXP20U#YWC0161#Einf_Aufzaehlen_T,
  3 updateBit    bit(01),
  3 einfZaehler  bin(31),
  3 pwbpoint     ptr,    /* session pointer */
  3 pltm         ptr;    /* session pointer */
```

Listing 5.2 EINF_AUFZAEHLEN callback input structure; contains all necessary variables and a bit indicating whether the update should take place.

Some aspects of the PoC implementation deviated slightly from the pattern. For instance, there was no "everything" mode because the procedure does not belong to any call hierarchy that is not asynchronous. It is only invoked from WC0161. The second change was that the procedure was duplicated rather than taken out of YWC0161 completely. The original procedure was then modified to perform a simple read rather than GHU and did not proceed with any of the update-relevant code. On the other hand, the new externally accessible callback version does not contain any code related to the write-only variables. A later reflection on the quality of this solution motivated the final approach in the pattern that avoids code duplication.

Let us now examine the application locking requirements. Table 5.1 shows the two lock candidates. It was discovered that the trading order from P90 in fact already has both a modify ticket and an exclusive locking mechanism. The interaction diagram for the application locking solution on P90 was worked out by the XPP Brokerage team and is included in Figure 9.4. The second lock candidate is the postponed update from module YWC0161. Interestingly the only variable that is being modified depending on the read data is VOLL_EINFZAHLER_BIT (see Listing 9.1). Hence it is only necessary to make sure that the GHU read during the postponed update leads to the same value of VOLL_EINFZAHLER_BIT as the initial dirty read. In other words, VOLL_EINFZAHLER_BIT is used instead of a modify ticket. To implement this simplified locking version, both the initial and modified values of VOLL_EINFZAHLER_BIT were added to the retention structure from Listing 5.2. The
initial value was used to initialize the variable in the callback and the new one was compared with the newly computed value. A mismatch results in an error print statement.

Pattern 4.6 "Base Variables in Storage Areas" was executed exactly as prescribed. It provided a basis for the implementation of a simplified version of patterns 4.7 "Save State" and 4.8 "Save Pointer Targets of Known Types". The idea was to leave the state including pointer targets untouched in one IMS region. The solution does not allow concurrency and relies on the process remaining in the region until the response messages arrive. This simpler version is not usable in a productive environment but suffices to make the program run-able. The implementation was trivial. An array was declared in WC0161 which would hold pointers to data areas. There was no external storage module, but all functions from Table 4.2 were created directly in WC0161. The INIT function allocates a new working area, SAVE saves a pointer to the area in the first free slot in the array and returns the array index as a correlation token. Function RESTORE return sets the pointer to the primary working area to the pointer at the requested index in the array. Finally, FREE was created to free the working area in one atomic operation. Note that the dynamically allocated data does not have to be freed explicitly. This is because in order to prevent memory leaks, the program developers had to invoke the deallocating procedures already when the program was originally created. In this respect, the requirements have not changed.

Pattern 4.11 "Line Callout Paths with Entry Points" was likewise implemented exactly as prescribed. No problems were encountered during the implementation.

The remaining pattern, 4.1 "Launch Remote Notifications" was omitted in the PoC because it does not fit into the simplified infrastructure displayed in Figure 5.2. The pattern expects the callout and notification calls to be performed directly by the callout module, where this natural flow has been sacrificed for the sake of simplicity. However, the pattern is structurally highly similar to 4.5 "Launch Late Updates from the Main Module". The derived expectation is that they both perform equally well.
6 Conclusion

This work addressed the problem of transforming a local subroutine call in an existing PL/I application into an asynchronous remote call. The more concrete goal was to transform the program into the Request/Callback pattern. This boils down to the task of separating the physical transaction into multiple consecutive ones that together form a logical whole, whereby the behavior of the original program is preserved. The technical requirement was that the different parts of the logical transaction are able to run in distinct processing regions.

The problem was deconstructed into independent challenges and a set of holistic approaches was built up from solutions to the individual challenges. The approaches were designed to form a spectrum ranging from a synchronous callout towards a fully asynchronous callout. The major features of the proposed solution are its cost-flexibility and fluidity which allow project leaders to establish an ideal target approach. At the same time, they account for the possibility of forces such as time pressure or new requirements that would compel the leaders to adapt less or more complete approaches, respectively.

The simplest conceptual form of asynchronous callout was additionally supported with a set of descriptive technical reengineering patterns. The patterns are primarily driven by ease of implementation, which means making well-defined modifications with only a minimal understanding of the underlying code. The process of gaining all necessary knowledge is also captured in reverse engineering patterns.

The lessons that follow from the detailed analysis and an accompanying proof of concept implementation are threefold. First, standardization of the transformation process makes the effort simple and predictable. Second, the required general applicability of the patterns means their results may be suboptimal. Finally, one can observe that the more detailed knowledge of a program’s dependencies a solution requires, the more difficult the resulting application is to maintain.

In conclusion, the presented conceptual solution has the potential to be used at Credit Suisse. While only future will show whether the current patterns with their high maintenance claims will suit the requirements of new projects, the modularity of the approach facilitates creation of new and improved patterns.
6.1 Future Work

The first peek into the problem revealed a lot of interesting challenges. Additionally, the proposed solution leaves plenty of room for exploration and improvement. Hence, there are many directions in which future work on this problem can be done.

First and foremost, to verify completeness of the approach, the PoC needs to go through a thorough testing phase. A performance comparison between the different stages of the asynchronous spectrum may also be performed to assist project leaders with decision making. On a more granular level, it is also worth exploring the effects of each pattern on the runtime of the program and making the results available in the patterns themselves.

The next step should be to complete the missing pieces of the puzzle, namely the testing and exception handling concepts. Modules which are universal should also be standardized – an example here is the storage module. Finally, a set of code generators can be created to assist or entirely replace some of the patterns.

As suggested above, the patterns should be revisited by experienced PL/I developers. They can make a valuable contribution by introducing new approaches, optimizations, ideas or tools. The quality of both the reengineering process and the resulting program can be significantly improved.

One may also examine completely different approaches and settings of this problem. While this work strived to achieve the Request/Callback pattern, it might be worthwhile exploring the alternative, which is a complete separation into three independent transactions. More specifically, the prologue, the remote transaction and the epilogue would be part of a chain of processing and not connected in any other way than by messages. The interesting question is what it would take to modify the original program to fit this model.

An additional direction to investigate may be the applicability of the developed concepts in a different environment, for instance the Java Application Platform. The prediction is that even though the patterns will have to be created from scratch, the general concepts may still apply.

Finally, it is definitely interesting to inspect the lessons that can be learned about development of new more flexible applications and likewise about the development of Request/Callback applications from scratch, i.e. the so-called greenfield approach.
7 References


<table>
<thead>
<tr>
<th></th>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XPP</td>
<td>eXtended Pricing Platform; Currently ongoing project at Credit Suisse.</td>
</tr>
<tr>
<td>2</td>
<td>Arcoba</td>
<td>Architecture for Core Banking</td>
</tr>
<tr>
<td>3</td>
<td>IMS</td>
<td>Information Management System; Database and transaction management system designed by IBM.</td>
</tr>
<tr>
<td>4</td>
<td>PL/I</td>
<td>Programming Language One; Language introduced in 1960s, nowadays predominantly considered a language of legacy applications.</td>
</tr>
<tr>
<td>5</td>
<td>JAP</td>
<td>Java Application Platform; Platform for Java programs in Credit Suisse.</td>
</tr>
<tr>
<td>6</td>
<td>callout</td>
<td>(Remote) call for which a response is expected. Credit Suisse term.</td>
</tr>
<tr>
<td>7</td>
<td>ACID</td>
<td>Transaction properties: atomicity, consistency, isolation, durability</td>
</tr>
<tr>
<td>8</td>
<td>PoC</td>
<td>Proof of Concept</td>
</tr>
<tr>
<td>9</td>
<td>BO</td>
<td>Business Object</td>
</tr>
<tr>
<td>10</td>
<td>PCB</td>
<td>Program Control Block; Definition of a program's access permissions.</td>
</tr>
<tr>
<td>11</td>
<td>GHU</td>
<td>Get Hold Unique; Database command required before updating on IMS databases</td>
</tr>
<tr>
<td>12</td>
<td>IMS session parameters</td>
<td>Pointers supplied to each program when it is loaded; they are session-specific; they are used to communicate with the transaction manager</td>
</tr>
<tr>
<td>13</td>
<td>puts</td>
<td>jargon for logging and other technical output operations</td>
</tr>
</tbody>
</table>
9 Appendix

9.1 Problem Analysis

Figure 9.1 Synchronous PoC infrastructure; source: [CD10A]
9.2 Reengineering Patterns

**If It Ain’t Broke, Don’t Fix It**

*The name is usually an action phrase.*

**Intent:** Save your reengineering effort for the parts of the system that will make a difference.

**Problem**
Which parts of a legacy system should you reengineer?

*This problem is difficult because:*
- Legacy software systems can be large and complex.
- Rewriting everything is expensive and risky.

*Yet, solving this problem is feasible because:*
- Reengineering is always driven by some concrete goals.

**Solution**
Only fix the parts that are “broken” — that can no longer be adapted to planned changes.

**Tradeoffs**

*Pros* You don’t waste your time fixing things that are not only your critical path.

*Cons* Delaying repairs that do not seem critical may cost you more in the long run.

*Difficulties* It can be hard to determine what is “broken”.

**Rationale**
There may well be parts of the legacy system that are ugly, but work well and do not pose any significant maintenance effort. If these components can be isolated and wrapped, it may never be necessary to replace them.

**Known Uses**

**Related Patterns**
Be sure to Fix Problems, Not Symptoms.

**What Next**
Consider starting with the Most Valuable First.

*Figure 9.2 Example of pattern structure; complete figure taken from [DS08]*.
9.3 Reverse Engineering Artifacts

Figure 9.3 Relevant reachable updates from YWBTAR before the callout; WSA are the trading order records extracted from P90.
9.4 Forward Engineering Artifacts

EINF_AUFZAehLEN: PROC(LAND,AART,TITEL);

DCL 1 SEG_DX,
   %INCLUDE DX2W;;
DCL DXPTR   POINTER;
DXPTR = ADDR(SEG_DX);

DCL PDB   POINTER;
DCL LAND  CHAR(2);
DCL AART  CHAR(5);
DCL TITEL CHAR(8);
DCL PIC2  PIC'99';;
DCL PIC1  PIC'9';
DCL DX_ZAEH  BIN FIXED(31);

DCL 1 QUAL_SSA,
   5 SEGNAME  CHAR (8) INIT ('WB2W    '),
   5 COMMAND,
   10 STERN   CHAR (1) INIT ('**'),
   10 CODE    CHAR (4) INIT ('----'),
   5 QUALIFIKATION,
   10 AKLAMM  CHAR (1) INIT ('('),
   10 Q_1,
   15 VERFELD CHAR (8) INIT ('WB2W04  '),
   15 VEROPERAND CHAR (2) INIT ('= '),
   15 VERWERT  CHAR (40) INIT ('DX0002 '),
   10 EKLAMM  CHAR (1) INIT (')');

PIC1 = 0;
PIC2 = 0;

IF AART = 'KUNDE'
THEN EINF_STOP_TXT = 'STOP WEGEN EINFÜHRUNG! TYP: KUNDE,'
   !! LAND !! ', ' !! TITEL;
ELSE EINF_STOP_TXT = 'STOP WEGEN EINFÜHRUNG! TYP: KONFORM,'
   !! LAND !! ', ' !! TITEL;

PDB = WBP2W;
CALL PLITDLI(CT4,'GHU ',PDB,DXPTR,QUAL_SSA);
IF PDB->TPSTC ^='' THEN RETURN;

DO DX_ZAEH = 1 TO HBOUND(DY2W07D,1)
  WHILE (DX2W07K(DX_ZAEH) ^= 'EF' & DX2W07S(DX_ZAEH) ^= 'A');
END;

IF DX_ZAEH <= HBOUND(DY2W07D,1)
THEN DO;
  IF EINF_ZAEHLER = 50
  THEN DO;
    PIC1 = SUBSTR(DX2W07D(DX_ZAEH),EINF_ZAEHLER,1);
    PIC1 = PIC1 + 1;
    SUBSTR(DX2W07D(DX_ZAEH),EINF_ZAEHLER,1) = PIC1;
    PUT SKIP LIST('PIC1'!!PIC1);
  END;
  ELSE DO;
    PIC2 = SUBSTR(DX2W07D(DX_ZAEH),EINF_ZAEHLER,2);
    PIC2 = PIC2 + 1;
    SUBSTR(DX2W07D(DX_ZAEH),EINF_ZAEHLER,2) = PIC2;
    PUT SKIP LIST('PIC2'!!PIC2);
  END;

VOLL_EINFZAEHLER_BIT = ON;

CALL PLITDLI(CT3,REPL,PDB,DXPTR);
IF PDB->TPSTC ^='' THEN RETURN;
END;

END EINF_AUFZAEHLEN;

Listing 9.1 Pre-callout updating code which had to be postponed; variables declared outside the procedure are highlighted in blue.
Figure 9.4 Locking mechanism on the trading order from P90; taken from [CD10B].