Reusing Legacy:
Models, Data and Systems

Author: Richard Veryard
Version:February 12th 1999

richard@veryard.com
http://www.veryard.com

For more information about SCIPIO, please contact the SCIPIO Consortium.

info@scipio.org
http://www.scipio.org

Preface

Purpose of document

Describes the techniques used by SCIPIO for extracting value from legacy models, legacy data and legacy systems.

Should be read in conjunction with one or more documented case studies for SCIPIO.

Assumptions

The reader should be familiar with model-based development, such as Information Engineering.

Outstanding questions

This document has been developed as a draft for discussion. Some open questions are interspersed with the text.

Q	Do you have any experience or opinions relevant to the question?
Q	Do you have any additional ideas or issues?

Acknowledgements

Introduction

Purpose

This document assumes that you want to extract components from your legacy application systems.

There are several reasons why we might wish to do this.

- Because we believe that there is (or will be) demand for this component, either internally or externally.
- Because we wish to add functionality to a legacy system using components.
- Because we wish to create interfaces between legacy systems and new systems.
- Because there is a specific problem with the legacy systems, and we want to use components as part of the solution. (One example is the use of bridging components to deal with the Year 2000 problem - see references.)
- Because we want the longer-term benefits of having all systems constructed from components.

Goals

We want to make maximum reuse of legacy assets, while minimizing the inheritance of legacy liabilities and constraints.

Legacy assets

> **Q** Where is the value in reusing legacy systems? How can we quantify the value of a legacy asset?

If an application system is currently in use within an organization, there must be a reasonable degree of fit between the system and the organization. The fit may be imperfect, and the users may have to undergo various inconveniences and tribulations to make the system work, but at least it does work. Furthermore, the users are familiar with the system, warts and all.

To the extent that the system works, it must contain some knowledge about the business requirements. The system has a reasonably good internal model; this model may be implicit or explicit, depending on how the system was developed and documented, and how it has been maintained.

Of course, the value of a legacy system does not necessarily bear any relation to the amount of time and effort you have put into it.

Legacy liabilities

> **Q** What limits the successful reuse of legacy systems? How can we quantify the depreciation of a legacy asset?

A legacy system may not work all the time. It may have undocumented features or erratic behaviour, especially when exception conditions occur. It may have bugs or poor performance.

A legacy system may have in-built constraints deriving from the way it was conceptualized or designed. The internal model may be adequate for a limited range of situations, but it may not be adequate for all situations. The design may make simplifying assumptions that limit future reuse.

For example, a legacy system may contain a batch procedure that is designed for single-threaded operation on a flat file. It would take a clever programmer indeed to wrap that procedure so that it would support multi-threaded operation.

A legacy system will usually be designed for operation on a specific technical platform. Considerable effort may be required to eliminate the dependency of the code on this platform, and to make it work on other platforms.

Furthermore, any poorly executed action on a legacy system, including wrapping, may worsen its quality and performance. Wrapping almost always introduces some performance overhead, and this is not always balanced by the technical superiority of the target plaftorm: sometimes a poorly wrapped system can run slower on a high-performance machine than it did in its original technical environment.

System Conversion

A common technical challenge with legacy systems is to convert them to some other platform, or to make them compliant with some new requirement. Converting legacy systems to support Year 2000 and Euro trading falls into this category.

A chunk of legacy system makes a component. Each chunk needs to be tested and implemented separately. Clean interfaces need to be defined between the chunks, so that converted chunks can interoperate with unconverted chunks. Management will be reassured when they see a growing number of converted chunks successfully tested and incorporated into production systems.

As each chunk of legacy system is converted, it is provided with two parallel interfaces: one for communicating with unconverted chunks, and one for communicating with converted chunks. This is achieved with 'intelligent' bridging components.

The alternative is a very high-risk strategy: big bang conversion, where the entire legacy portfolio is converted from non-compliance to compliance in a single overnight integration test and implementation.

For many conversions, such as Year 2000 conversions, are regarded as simply a programming task: to find all the date references in the code and fix them. Managers argue that any higher level analysis is both a distraction (given the urgency of the task) and an impossibility (given the poor structure of the systems and the unreliability, incomprehensibility or sheer absence of documentation).

But most people would accept the need for some form of testing before the altered programs are returned into the production environment. This testing needs to include system/integration testing - does this program still work in conjunction with other programs - as well as unit testing.

But here's the difficulty. If you don't know what a system is supposed to do, you can't test it. The ability to conduct a meaningful test implies a specification of the system. And if you don't know what separate subsystems are supposed to do, you can only test the whole system as one large lump. In such circumstances, debugging is a hit-and-miss affair, as likely to add bugs as to remove them.

Therefore, if your legacy systems are poorly structured and lack good documentation, some form of analysis is all the more necessary. The analysis we recommend concentrates on creating a relatively small number of large components with a small number of access points into each one.

Overall Task Structure

This task structure is extracted from the SCIPIO development process framework.

Application Assessment

- Assess Demand for Component
- Assess Current Application Quality
  - Review Current Application against Objectives
  - Establish Application Benchmark
  - Set Targets for Application Improvement
  - Explain Application Characteristics
  - Estimate Characteristics of Solution (simulation)
  - Review Characteristics of Solution (post-implementation evaluation)

Application Analysis

- Map Current Application
- Document Application Components
- Document Application Interfaces and Protocols
- Document Application Responsibilities
- Map Process to Application

Application Design

- Information Sourcing
- Component Identification
- Component Protocol Design
- Component Responsibility Assignment
- Data Storage Design
- Component Specification
- Component Sourcing

Application Conversion

- Transition Component Construction
- Component Publication
- Solution Testing
- Solution Deployment

Assessment

Assess demand

If the primary purpose of the project is to satisfy some demand, either internal or external, then the focus of this stage will be to assess this demand.

If our intention is to sell components outside the organization, then we need to have a reasonable idea of the revenue we could expect to earn. We need to have some way of assessing the number of organizations or individual users that might be expected to use a given component, and the price that could be charged. If we expect other competing components to be available in the marketplace, we should estimate the market share that we might reasonably achieve.

Any organization that intends to build components for commercial sale should have a development policy that limits the development expenditure to a certain proportion of the expected revenue.

Development	30%
Documentation	10%
Publication / Marketing	30%
Support	10%
Profit / Risk	20%

Figure 1: Typical commercial development policy

Even for internal reuse, we need to make some judgement of the likely level of reuse, so that we can allocate scarce development resources to things that have the greatest reuse value.

An assessment of competing components will also help us create a component that emphasises the unique features and benefits that we are able to offer.

Assess application quality

Often the primary reason to convert legacy systems into components is to improve the quality of the system itself, or to create new interfaces into the system. This requirement may form part of a larger development or maintenance project using Component-Based Development.

In this situation, the focus of assessment will be the quality of the legacy system itself, in terms of its functionality, maintainability, user-friendliness, efficiency, reliability and portability. This assessment will guide us towards those parts/aspects of the system where we are likely to get the greatest benefits of conversion to components. For example, if one portion of the legacy system has a history of frequent change, this will be a candidate for early conversion.

Analysis and Design

System archaeology and dissection

Legacy code is often compared to spaghetti, but lovers of Italian food will recognize that pizza provides a much better analogy. You try to cut out a wedge of pizza, but it remains connected to the rest of the pizza by innumerable strands of elastic cheese.

Or we might prefer to compare legacy code with lasagne. What were once separate layers of cheese, pasta and other ingredients have now been melted into a solid mass.

When we look at the legacy systems of a large company, we are usually presented with what appears to be a list of application systems. But this list can be misleading. The items in the list refer to the original system development projects - the layers of the lasagne. The systems have usually grown and changed, in functionality and architecture, often to the point where the original names no longer seem appropriate. New data stores, or interfaces to remote data stores, may have been added adhoc.

Even if the applications were originally designed according to a well-thought-out architecture, with maximum cohesion and minimum coupling, evolution of the applications over time may have greatly reduced the cohesion within each application and increased the coupling between applications.

So the segmentation of legacy code may not follow the apparent boundaries between the legacy applications. We need to identify ways of carving up the code with maximum cohesion and minimum coupling. Sometimes you can improve the situation by reducing connectivity between subsystems.

The idea is to convert an interlocking portfolio into a manageable set of segments connected by defined interfaces with bridges at each crossing point.

We may identify different design patterns in legacy systems.

Different design approaches focus on different types of simplicity and coherence in the designed system. Within traditional structured methods, such as Information Engineering, several design patterns are possible. We can identify two main patterns here: the data-driven approach and the event-driven approach.

Of course, many legacy systems have been extensively altered since their original design conception. This may mean that the original patterns have been largely lost or heavily compromised. But it is usually possible to find some parts of the system where the original pattern can still be seen.

The reason we are interested in identifying the patterns of the original design is that this tells us what kind of components we should expect to find buried in the legacy system. Data-driven design patterns tends to yield data-oriented components, while event-driven or process-driven design tends to yield business-service or process-oriented components.

> **Q** Should we identify any other standard patterns of legacy system design?

Data-driven design

The designer of the legacy system may have adopted a data-driven approach: focusing on achieving simplicity of data access. Such a system offers simple data access for each data object, based on a standard pattern:

C: Create

R: Read

U: Update

D: Delete

The overall coherence of the system is again focused on data structure and access. Each computer transaction acts on one data object, or closely related data objects. Typically, the same screen or window design would be used for all data accesses on that object. Sometimes, this pattern will be complicated by the need for different users to have different levels of access to the same data objects. (In other words, some users are allowed to perform create/update operations on a given data object, while other users are only allowed to perform read operations.)

Some development environments allow for stored procedures to be attached to data objects in the database. When these data objects are wrapped as components, these procedures normally become part of the component behaviour.

With data-driven legacy systems, it is usually relatively easy to identify and isolate components offering data services.

> **Q** Should a data access component contain its own accessing rules? How would this be implemented? What are the other options?

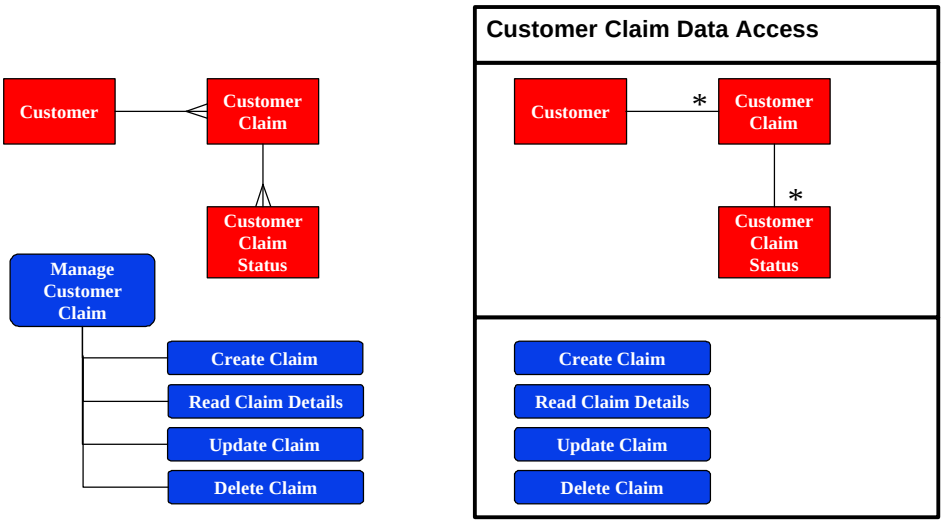


Figure 2: Data-driven design yields data access component.

Event-driven design

Alternatively, the designer of the legacy system may have adopted an event-driven approach: focusing on achieving simplicity of processing business events. In such a system, each type of business event triggers a separate computer transaction.

The designer focuses attention on the coherence of events and object life-cycles. Event coherence is achieved by completely handling each event by a single transaction. An event triggers state transitions of one or more data objects, and the life-cycles of these data objects should be complete and consistent.

Event-driven legacy systems may yield components offering process-oriented services.

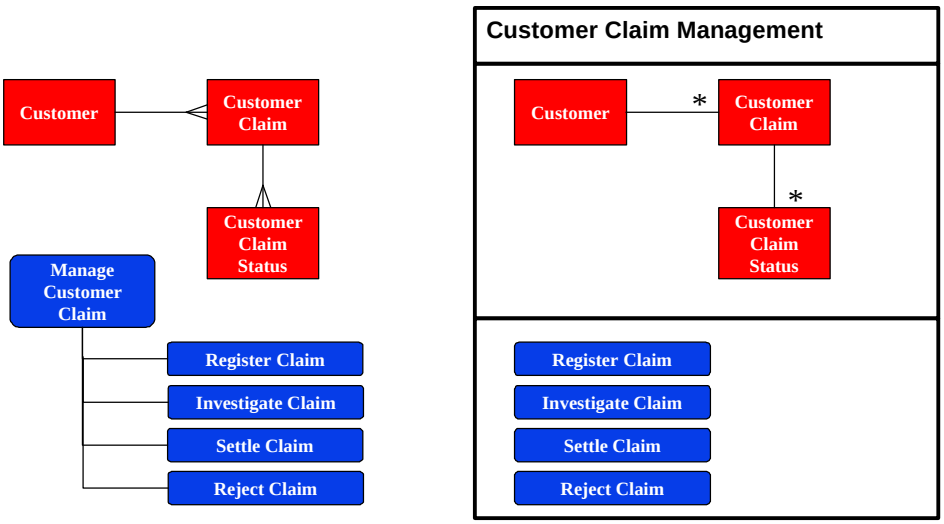


Figure 3: Event-driven design yields business service component.

Hybrid design

Many legacy systems contain some subsystems that are predominantly data-oriented and some subsystems that are predominantly event-oriented.

This may either be part of the original design concept for the system, or may be a result of later additions and modifications.

Data Structure

Data subtyping

Traditional data modelling

Using traditional data modelling, data objects are defined according to a common structure of attributes and relationships. Thus CUSTOMER ARCHIVE is regarded as the same data object as CUSTOMER, because it has the same attributes and relationships. However, CUSTOMER ARCHIVE has quite different behaviour to CUSTOMER, and it is required by a quite different set of procedure objects. The behaviour of CUSTOMER ARCHIVE probably has more in common with that of PRODUCT ARCHIVE and EMPLOYEE ARCHIVE.

Traditional entity modelling allowed for subtypes to be identified, but these were often clumsy to implement on relational databases. For this reason, many legacy data models don't contain subtypes.

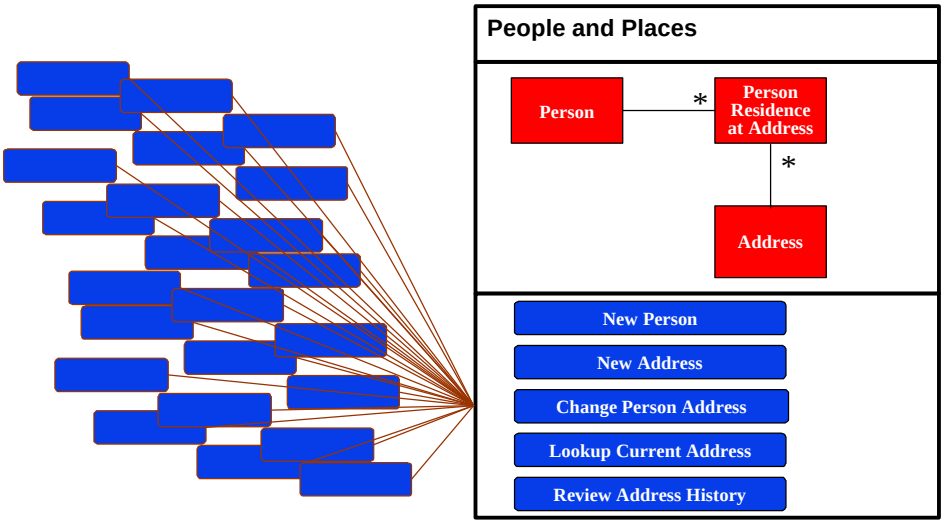
Component-based development

Object orientation contains the notion of **multiple inheritance**, which essentially means that a data object can be a subtype of many different supertypes. Thus CUSTOMER ARCHIVE may inherit its data structure from CUSTOMER but it inherits its behaviour from a generic procedure object called ARCHIVE. This would include such operations as placing things in the archive, accessing archive data, and restoring data from archive.

There are various ways this can be implemented through components. For example, if the archive behaviour can be suitably generalized, it can be wrapped into a component that is called by anything that requires archiving.

Alternative data views

In many traditional database systems, every program must use the same data structure.



In this example, we have a complex data object, which is composed of three simpler data objects.

If we build a single component to provide all data access to this compound data object, this has the advantage of consistency. Any changes to the data structure are made in one place only.

However, this solution has two significant disadvantages: performance and maintenance.

- In some technical environments, such a component may become a performance bottleneck.
- Any changes to the component interface will require changes to all the programs and components that use this interface.
- In some cases, depending on technical environment and management policy, all changes to the component may require some reconstruction and retesting of all the programs and

components that use this interface, even if these changes are internal to the component and don't alter the interface.

In component-based systems, some programs may use a different data service.

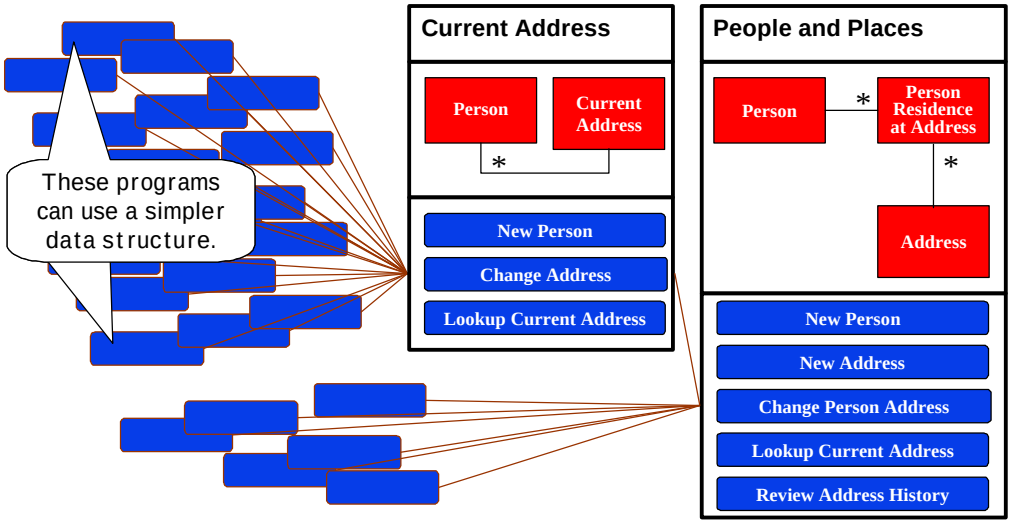


Figure 4: Hiding data structure.

What we would like to do is define a simplified view on the data, which most of the users can access rather than having to handle the full complexity of the data structure. Most relational databases support such views (sometimes called subschemas). However, many CASE tools do not use this RDBMS feature, and require all procedures to access the logical data model directly.

If we define a second data access component to support this simplified view, we can simplify most of the programs by switching them over to this view. Consistency is maintained, because both data access components are supported by the same persistent data storage.

Note that we don't have to switch all the programs over at once. We may wait until there is a specific maintenance requirement on a program, or convert a few at a time according to available resources.

The more data service components we introduce, the easier it gets to change the data structure.

The more programs we can switch over to these data access components, the easier it gets to change the data structure. In Figure 5, only those programs that use the full PERSISTENT DATA STORAGE interface may require rework, recompilation, retesting and/or reinstallation.

Note that Figure 5 shows the logical component interfaces, rather than the physical packaging of components. In technical environments where it is possible to build components with several different interfaces, the designer may choose to package both the DATA SERVICE 1 interface and the PERSISTENT DATA STORAGE interface into a single physical component. However, the same argument applies.

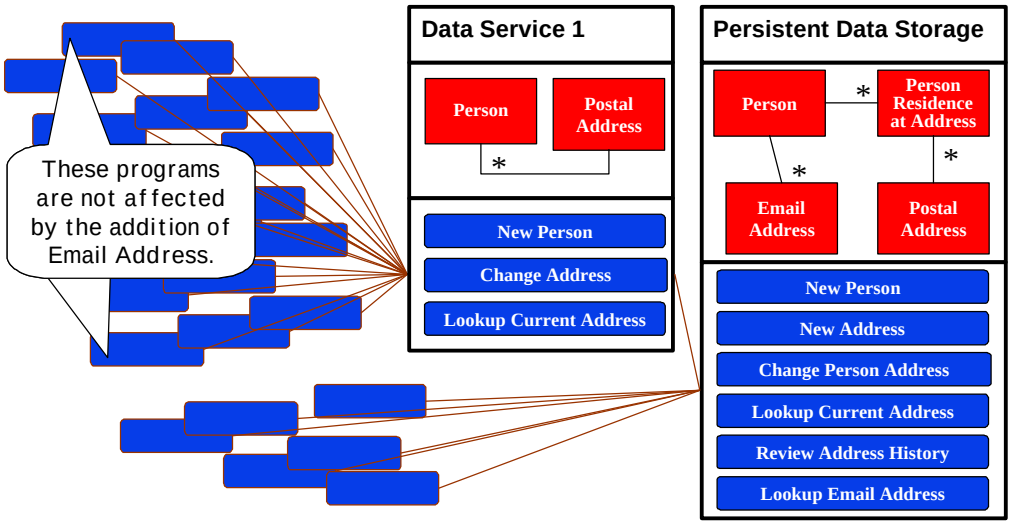


Figure 5: Hiding changes to data structure.

If we introduce business service components, we can put business logic in one place.

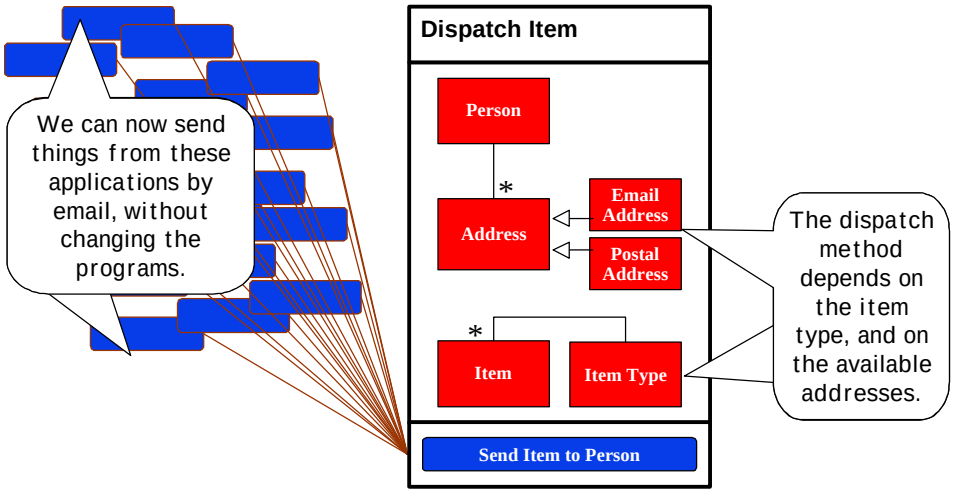


Figure 6: Hiding business logic.

Furthermore, we can use this approach to take some of the logic out of the programs. In the example shown in Figure 6, we have encapsulated the choice of dispatch method within the DISPATCH ITEM component.

Clustering

A legacy system can be seen as one very large component.

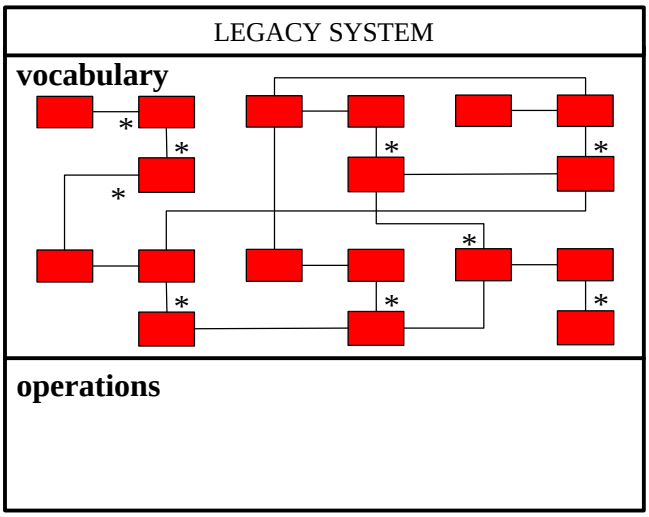


Figure 7: Legacy system as class.

All legacy systems perform a set of operations, which define its behaviour. In order for these operations to work at all, the system needs to have a vocabulary (or implied data structure).

If the legacy system has been developed and maintained using models, such as data and process models, these models should provide a good first cut of the vocabulary and behaviour of the system.

In some cases, however, the system models will include data and procedure objects that have been introduced for internal design and implementation purposes only. These should not be regarded as part of the business requirements.

We identify clusters in the legacy systems.

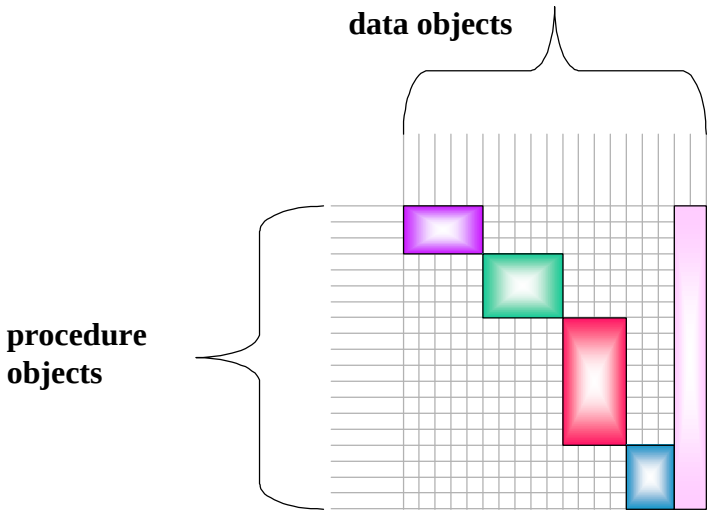


Figure 8: Interaction clustering.

Information Engineering includes techniques for analysing the interactions between data objects (usually entity types) and procedure objects (usually either elementary processes or procedure steps). The interactions, which may be Create, Read, Update or Delete, are shown on a matrix colloquially known as a CRUD matrix.

Based on these interactions, the rows and columns of the CRUD matrix are shuffled until the interactions roughly form rectangular blocks. This technique is known as clustering, and the resultant blocks are known as clusters. Some modelling tools perform an initial clustering automatically, although the results usually need some intelligent interpretation and adjustment.

These matrixes are used within Information Engineering to support top-down analysis - at each stage, the clusters define the scope of several areas to be developed further in the following stage.

A similar technique is used in analysing the structure of a legacy system.

If a clustering technique was used in the original development of the legacy system, we should normally expect to find these clusters still existing. However, this is not always the case, as subsequent maintenance activity may have blurred or shifted the boundaries between the original clusters. Therefore, if the legacy system has been subject to a significant amount of maintenance activity, it is worth confirming the clusters before proceeding.

We can simplify the interactions in various ways.

If the clustered matrix is too complicated, there are various ways of simplifying it.

If a data object is used by a large number of procedure objects, look for ways of partitioning the data object.

By attribute	Do some of the procedure objects only need some of the attributes of the data object?	The data object CUSTOMER ACCOUNT provides access to a complete transaction history for this customer, but many procedure objects only need the attribute CUSTOMER ACCOUNT BALANCE.
By occurrence	Do some of the procedure objects only need some of the occurrences of the data object?	Procedures relating to the payment of sales commission refer to a subtype of EMPLOYEE, which we may call SALESPERSON.
By life-cycle stage	Do some of the procedure objects only need part of the life-cycle of the data object?	CUSTOMER PROSPECT is used by a different set of procedure objects to CUSTOMER.

We analyse the interactions between clusters.

We start from the assumption that each of the clusters we have identified in the legacy system is a candidate for conversion into a proper component.

Having identified the clusters, we look at the CRUD interactions that fall outside the cluster blocks in the clustered matrix. These represent interactions between the candidate components.

We can then draw an interaction diagram showing the interactions between these clusters. (Thus we regard the legacy system as a collaboration between clusters.) Figure 9 shows an exchange diagram that corresponds to the clustered matrix in Figure 8. With further analysis, we could draw a UML-style sequence diagram or collaboration diagram, although this is often unnecessary.

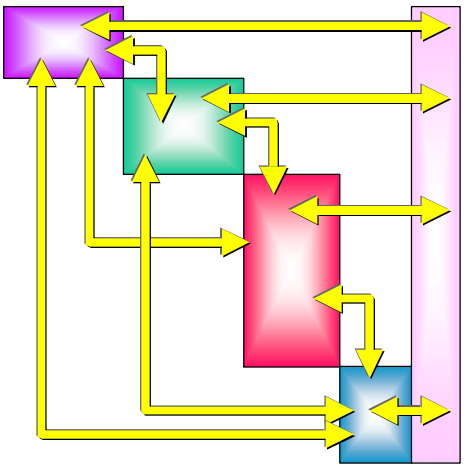


Figure 9: Interactions between clusters.

We analyse the business rules that belong to each cluster.

Figure 10 shows a project assignment system, which we have analysed as a collaboration between three clusters: PROJECT, EMPLOYEE and ASSIGNMENT. We might expect this to fall naturally into three components.

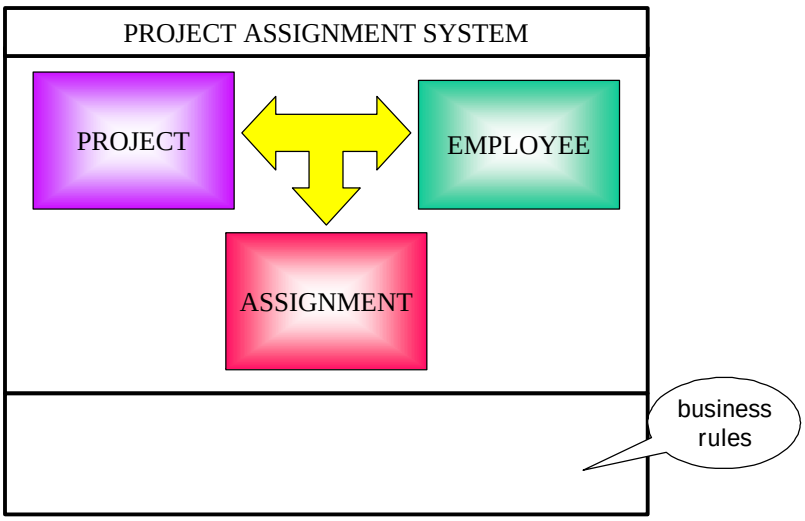


Figure 10: Legacy system drawn as single collaboration.

However, when we look at the business rules that are contained within the system, we find they fall into four categories.

- Business rules relating to employees alone.
- Business rules relating to projects alone.
- Business rules relating to employees and assignments.

- Business rules relating to projects and assignments.

There are no business rules relating to assignments in isolation. Indeed, assignments themselves have no meaning in isolation.

In situations where the business rules are many and/or complex, it is useful to draw a rule hierarchy diagram. Sometimes it emerges from this analysis that the legacy system has implemented the business rules inefficiently or inconsistently, or that there are apparent loopholes in the system. This creates an opportunity to correct the business rules (with appropriate validation and verification involving business users and domain experts) before allocating the rules to the components.

Looking at the business rules would lead to an alternative way of dividing up the legacy system into two components: PROJECT ASSIGNMENT and EMPLOYEE ASSIGNMENT, as shown in Figure 11. Each business rule is allocated to one of these two components. The interface between the two components is concerned with maintaining a consistent link between ASSIGNMENT within PROJECT ASSIGNMENT, and ASSIGNMENT within EMPLOYEE ASSIGNMENT.

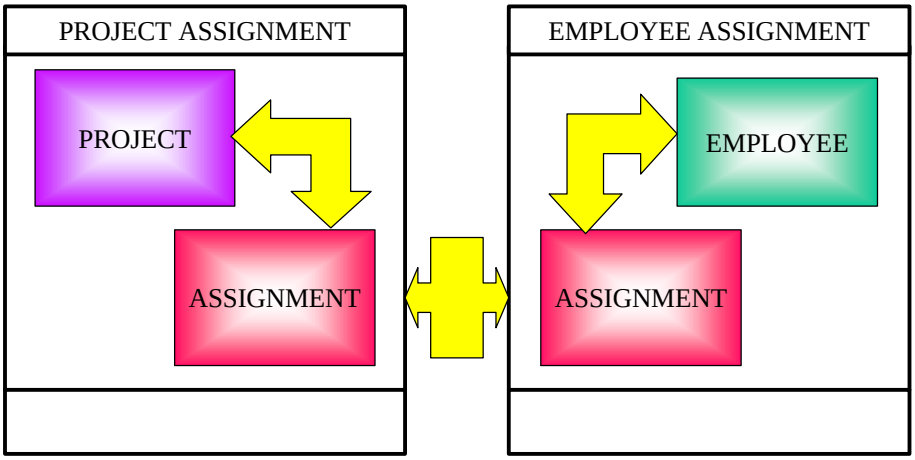


Figure 11: Legacy system divided into two collaborating subsystems.

Another advantage of this approach is that it focuses the creation of components on the project-assignment relationship and the employee-assignment relationship. If these two relationships can be generalized, we may be able to create components that will be suitable for a range of other situations. Indeed, at a suitable level of abstraction, we may be able to identify a significant amount of common behaviour between these two components.

Implementing the Component Structure

We implement the component structure in small steps.

Having analysed the legacy systems into potential components, we don't have to convert everything at once. One of the attractive features of component-based development is the ability to implement solutions in small steps. We may wrap and re-install one component at a time, if this suits our budgets and priorities. Or we may wait until an enhancement request affects a given area, and attempt to find a component-based solution that satisfies the enhancement request.

We implement components according to specific requirements and opportunities.

Component-Based Development

- New systems can be build as components.
- It's not necessary to enforce CBD on all development projects at once.
- New components and systems always connect via specified interfaces.
- Existing functionality needed by new systems should be converted into components.

Component-Based Maintenance

- Look at maintenance history and trends.
- Look at opportunities to hide data complexity or business logic inside special components.
- Each act of maintenance should aim to make the overall system MORE flexible.

References

SCIPIO Development Process Framework. Available on the web at http://www.scipio.org/

SCIPIO: CBD for Year 2000 and Euro. Available on the web at http://www.scipio.org/

