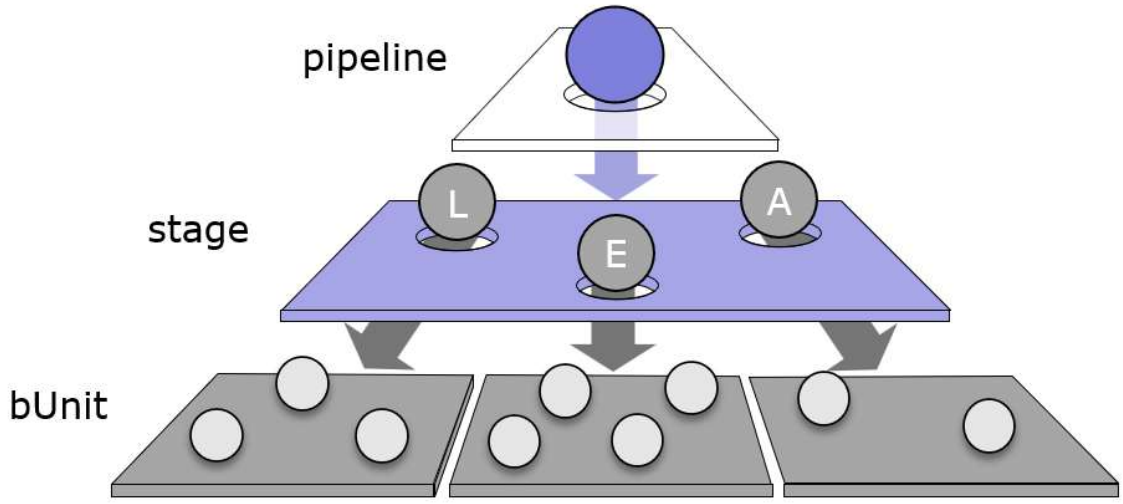
The bCLEARer Pipeline Architecture Framework eManual



As at 12th May 2025

Preface

BORO's experience shows that the ongoing evolution of radical innovative practices like bCLEARer depend upon adequate levels of inspectability. We have found that living documentation is a critical—though sometimes costly—enabler of this inspectability.

- At a day-to-day, hygiene level, we have found that adequate documentation reduces vagueness and inconsistency, streamlining knowledge-sharing and supporting smoother scaling.

- Strategically, we have found that it provides an informed foundation for the regular reflection that guides development. This helps us to clearly spot and assess successes and gaps, as well as avoiding dead ends.

A while ago, a client engaged us to crystallise our then current working documentation into an eManual for their bCLEARer programme. What follows is a sanitized version of that manual, capturing the architecture as it existed then. Though bCLEARer continues to evolve, the core principles in this eManual remain relevant.

We share this to support anyone running bCLEARer projects. Use it freely—it's offered as is.

Note

This document is a PDF rendition of an HTML eManual. For those interested in the original, it can be downloaded from here: https://borosolutions.net/bclearer-pipeline-architecture-framework-emanual

1 Getting Started

This bCLEARer Pipeline Architecture Framework Manual is one of a series of manuals that aim to get you started with bCLEARer.

The first section provides an introduction. bCLEARer - an introduction (see page 2) provides a general introduction to bCLEARer. Its sub-section - bCLEARer's Pipeline Architecture Framework - an introduction (see page 4) - gives an introduction to bCLEARer's pipeline architecture framework - the topic of this manual.

The subsequent sections - shown in the contents listed below - describe the framework.

Contents

Appendices

The manual contains these appendices:

1.1 bCLEARer - an introduction

1.1.1 bCLEARer

bCLEARer stands at the forefront of digital transformation, championing an evolutionary approach to harnessing digitization and digitalization opportunities. It guides information on a transformative journey, curating its evolution into fitter forms, ones more suited for computing, that deliver increased value.

To accomplish this, bCLEARer has evolved an architecture framework for semantic data pipelines, along with a methodology for engineering these pipelines. This manual explains the framework.

1.1.2 Its Digital Transformation Journey

This journey features two pivotal stages:

Digitization: Extracting and converting information from existing sources into accessible, computer-readable formats for further transformation.

Digitalization: Mining, extracting (making the implicit explicit), refining and evolving information to boost its utility and efficiency, enhancing its 'fitness' for digital ecosystems.

1.1.3 As Directed Evolution

The bCLEARer journey can be regarded as a form of directed evolution for information. One that mimics the processes of natural selection, steering information toward a fitter, improved state. This steering is enabled by the improved visibility not only of the transformations, but also the patterns of incoherence and inconsistency in the data. This transparency enables easy identification and correction of issues, fostering trust and reliability in the process.

1.1.4 Rapid, Radical, Resilient Evolutionary Changes

The bCLEARer journey typically involves rapid, often radical, (though resilient) evolutionary adaptations at scale. These occur simultaneously on two fronts:

Information Evolution: Adaptation of information throughout its journey.

Journey Evolution: Adaptation of the journey itself to emerging requirements, accelerating the information's evolution.

The whole process supports this rapid evolutionary adaptation: identifying and accommodating significant changes in both the information and its digital journey. A key element is adaptive resilience: maintaining stability and efficiency amidst continuous change.

1.1.5 Deployability and Adaptability

bCLEARer is deployable consistently and efficiently across a wide range of domains in programmes and projects of varying size and shape. This adaptability is underpinned by flexible, scalable, and maintainable processes.

1.1.6 Core Practices and Patterns

Over three decades, bCLEARer has established core practices and patterns to consistently meet the challenges of change and variety, ensuring effective implementation. (for more details on bCLEARer see BORO, BORO Foundation and bCLEARer --- A very brief introduction (see page 116)).

1.1.7 bCLEARer's Pipeline Architecture Framework - an introduction

The bCLEARer pipeline architecture, as part of the bCLEARer environment, has evolved along with it as a resilient system, also striking a good balance between stability and flexibility. The pipeline architecture core practices and patterns help to deliver the right level of flexible stability - providing the stability within which the system can flex. The architectural framework encapsulates these practices and patterns.

The pipeline architecture practices and patterns are vital to giving bCLEARer a firm foundation. These have been distilled into the bCLEARer Framework - Architecture - Pipeline (bCFAP). This report delves into the current bCFAP, highlighting its pivotal role in digital transformation.

This document describes the four primary elements of this framework (with more detailed material in the Appendices (see page 39)).

The initial three sections focus on the core aspects of the bCLEARer architecture:

1. **Pipeline Architecture:** The first section describes the pipeline (or pipe-and-filter) architecture a prevalent approach for data transformation. (Additional background material is in the appendix: Appendix - The Standard 'Pipeline' or 'Pipe-and-Filter' Architecture (see page 40)).
2. **Nested Gated Architecture:** The second section explores the nested gated architecture, which enhances scalability through its nesting structure and ensures process transparency through gating mechanisms.
3. **Three-Core-Level Nesting Architecture:** The third section examines the breakdown into broad three core level architecture of the nesting. In this, the mid-level is structured to reflect bCLEARer's five stages of evolution: Collect, Load, Evolve, Assimilate and Reuse (BORO, BORO Foundation and bCLEARer --- A very brief introduction (see page 116) gives more details of these). It also describes the two extended levels of nesting that can be added to the core levels.

The fourth, and concluding, section presents the general design practices and patterns adopted in the architecture.

Detail, including more technical matter and reference material, is relegated to Appendices (see page 39). This includes a glossary of the main terms (Appendix - Glossary of Major Terms (see page 47)) and a reference iconography (Appendix - Pipeline Architecture Iconography (see page 57)).

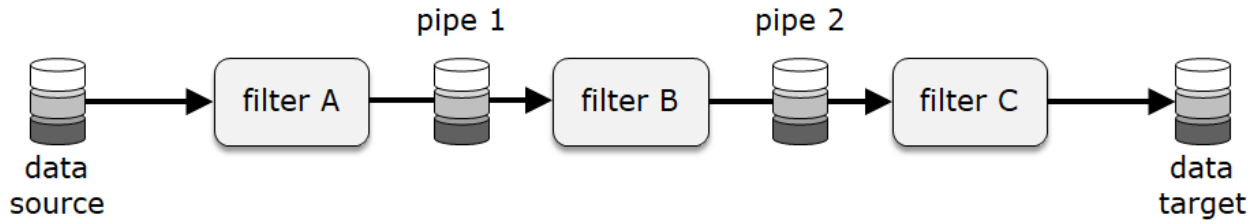
1.2 Pipeline or pipe-and-filter architecture

1.2.1 Introduction

As is often noted in the literature, data transformation systems typically have a ‘pipeline’ or ‘pipe-and-filter’ architecture (see, for example, the extract in Appendix - The Standard 'Pipeline' or 'Pipe-and-Filter' Architecture (see page 40)). ‘Pipe-and-filter' is the original name for this architecture, though ‘pipeline’ is a more common name nowadays. The bCLEARer approach is a data transformation process and so, unsurprisingly, is implemented with a pipeline architecture – giving rise to a bCLEARer pipeline. In this section, we look at the pipeline architecture, including how it can be nested. In the next section, we look at the pipeline architecture’s core hierarchical, nesting structure and how larger pipelines are typically given an extended, modular, nesting structure to facilitate management.

1.2.2 Visualising the pipeline flow

This architecture consists of a sequence of processing components, arranged so that the output of each component is the input of the next one creating a ‘flow’. A simple visualisation of a pipeline flow is given below:

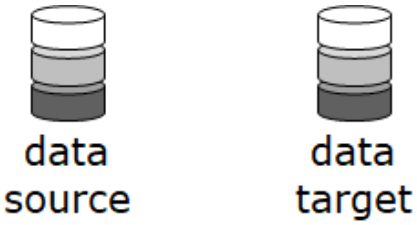


simple pipeline structure

The pipeline architecture has, as the 'pipe-and-filter' name suggests, a series of pipe and filter components, where pipes pass data to and from filters that transform the passed data — the pipeline flow. As well as the general inter-filter pipes, there are two specific types of pipes, the start and end pipes;

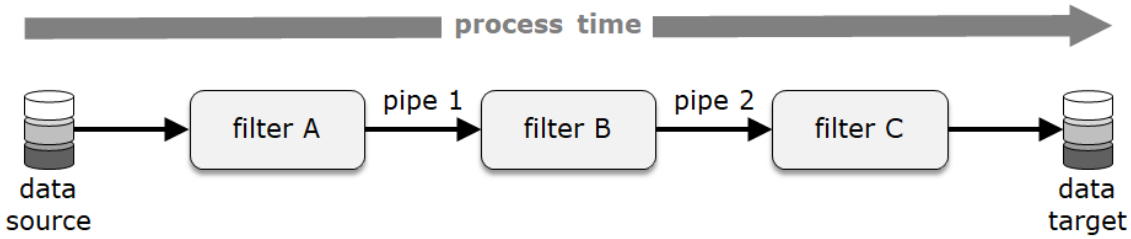
- **start:** a data source (pipe) to feed the pipeline and
- **end:** a data target (pipe) to persist the transformed data.

As illustrated in the figure above, these pipes are typically adorned with a dataset collection icon (as shown below).



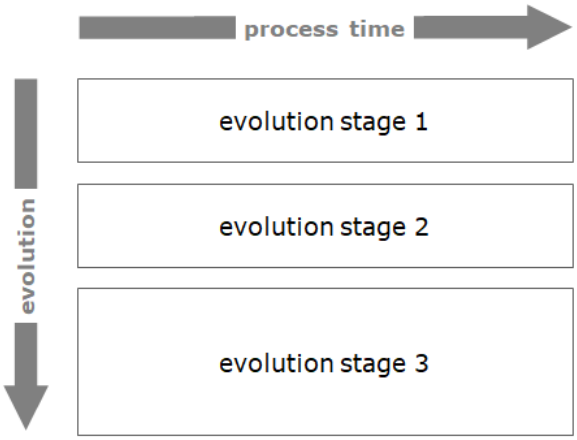
1.2.3 Process time

Conventionally, the pipeline flow - pipes passing data and filters transforming it - is shown across the diagram from left to right. This is made explicit in some diagrams with the use of a 'process time' arrow, as shown below.

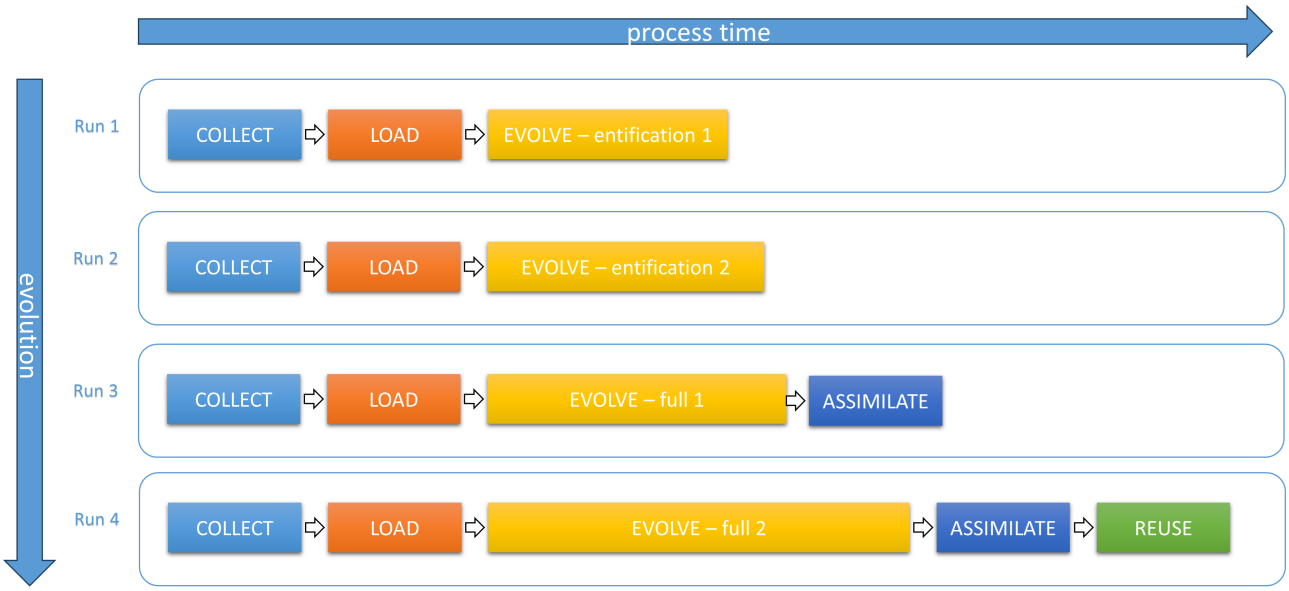


1.2.4 Evolutionary time

As noted earlier, the bCLEARer flow itself will typically evolve over time, sometimes radically, to accommodate emerging requirements. Pipes and filters may be added and removed - or changed. Sometimes one needs be able to represent this evolution in diagrams. This is typically represented as a series of snapshots of the pipeline (process) arranged down the page with an arrow from top to bottom showing the direction of evolution - a pro-forma example of this is in the figure below.



The simplest, ubiquitous kind of evolution is the iteration and extension of the bCLEARer stage level - shown below.



Similar simple iteration and extension will happen at the other two levels of nesting, as the project unfolds.

These are examples of run time evolution. There is also design time evolution. For examples of gate design evolution over time see figures in Nested pipeline's data stage gates (see page 18).

1.2.5 Pipeline components

The pipeline has two core components, 'filters' and 'pipes' (including the data source and data target pipes).

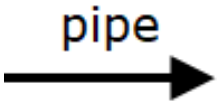
1.2.5.1 Filters

Filters are components that transform ('filter') data that is received as an input via pipe connectors. The icon for a filter is shown below:

filter icon

1.2.5.2 Pipes

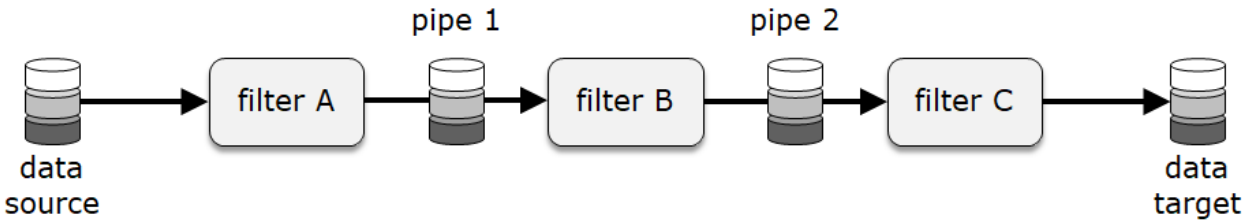
Pipes are the connectors for filters. The role of a pipe is to pass messages, or information, to and from filters. The flow is unidirectional, and, when needed in an implementation, the data is persisted until the filter processes it. The icon for a pipe is shown below:



pipe icon

1.2.5.2.1 Pipes adorned with data icon

As noted earlier, there is a data source pipe at the start of the pipeline to feed it and a data target pipe at the end of the pipeline to persist the transformed data. And these are usually adorned with a dataset collection icon. All pipes, not just the start and end pipes, transport data. Optionally, this point can be highlighted through the use of a data icon (in this case, the dataset collection icon) on any pipe in the diagram including those pipes inside the pipelines — as shown below for pipes 2 and 3.



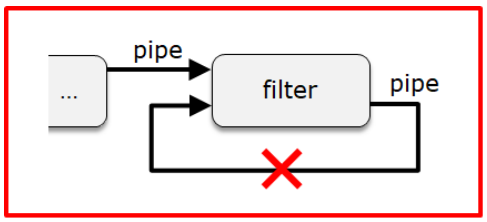
1.2.5.3 Filters and their pipes

Filters always have at least one input pipe and one output pipe, as shown in the following diagram:

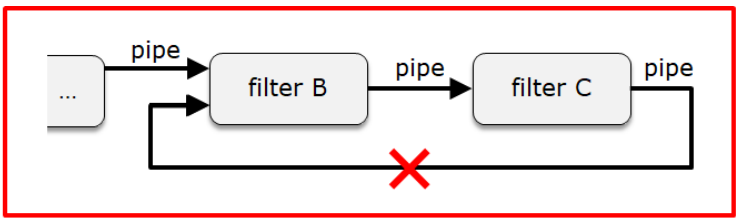
filter and its pipes

1.2.5.3.1 Acyclic pipeline flow

In this pipeline architecture, a filter's pipe cannot flow to itself, either directly or indirectly. The flow is acyclic — with no cycles. This does not inhibit reuse, as the same processing may be reused in different filters.



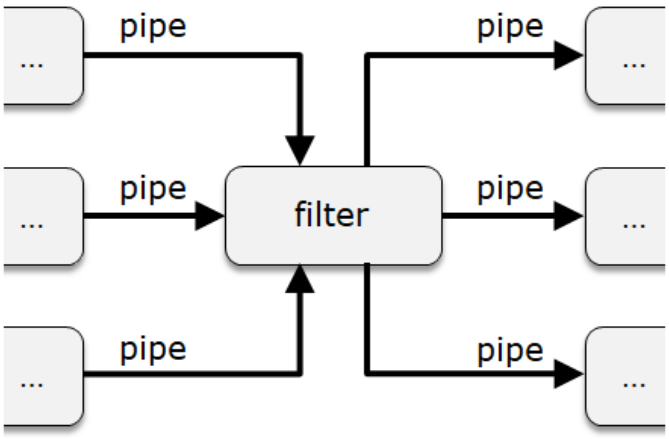
direct cycle (loop)



indirect cycle

1.2.5.3.2 Multiple pipes to and from different filters

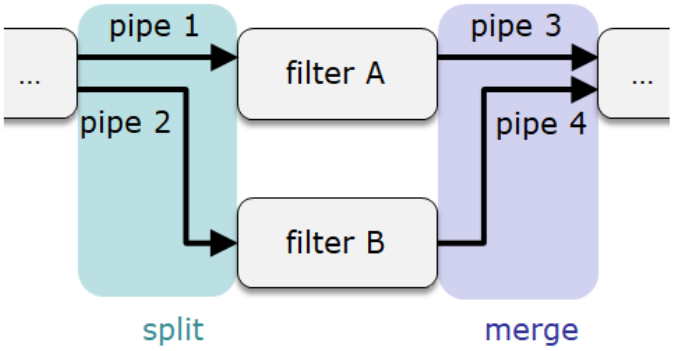
A filter can have several input pipes and several output pipes, as shown in the following diagram:



filter with multiple input and output pipes

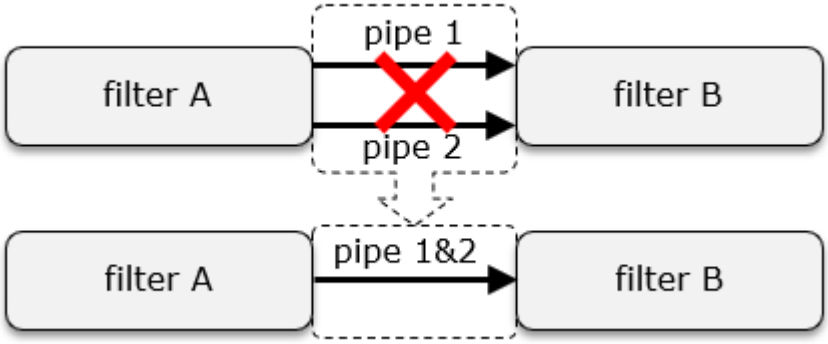
1.2.5.3.2.1 Merging and splitting pipelines

Filters with multiple input and output pipes can be organised into pipeline flows that split and then merge — as shown below.



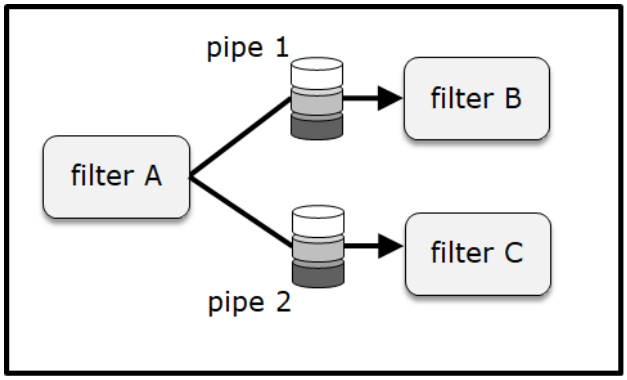
1.2.5.3.2.2 Multiple pipes between the same filters

None of a filter's pipes should flow to the same destination — where this happens, the pipes should be encapsulated — as shown in the diagram below.

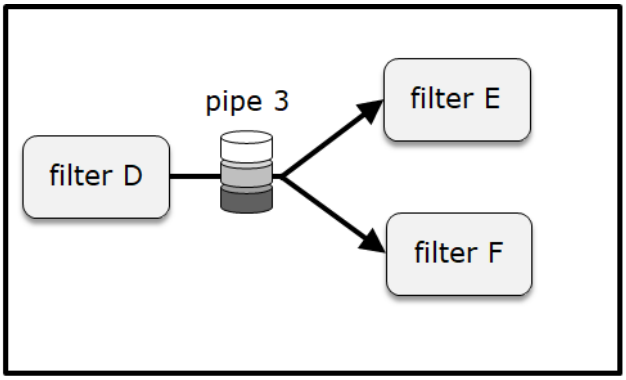


1.2.5.3.2.3 Pipes: single or multiple filter inputs

Typically a pipe will have a single filter output fed by a single filter input. However, there can be cases where it is important to record that the same data is fed from one filter to many other filters. In these cases, the pipe is shown with a single filter output feeding multiple filter inputs. Pro-forma examples of these two cases are in the figure below.



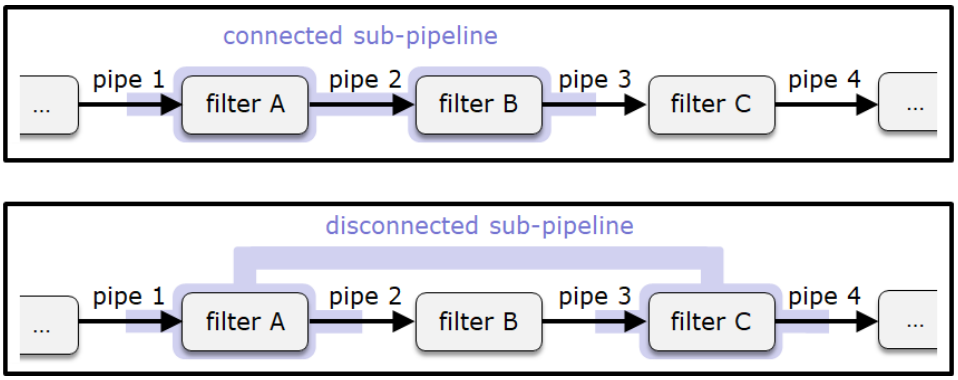
different data piped



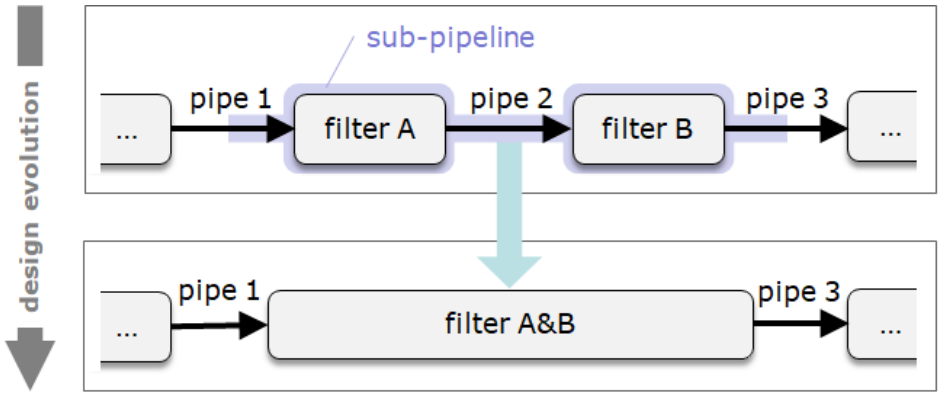
same data piped

1.2.6 Nesting

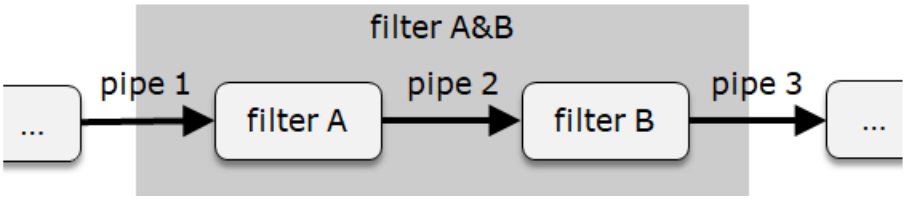
Any subset of filters in a pipeline forms a sub-pipeline, however, it can be useful to distinguish between pipelines that are connected and disconnected — see below.



This makes it simple to organise a sub-pipeline into a nested pipeline — as shown below.

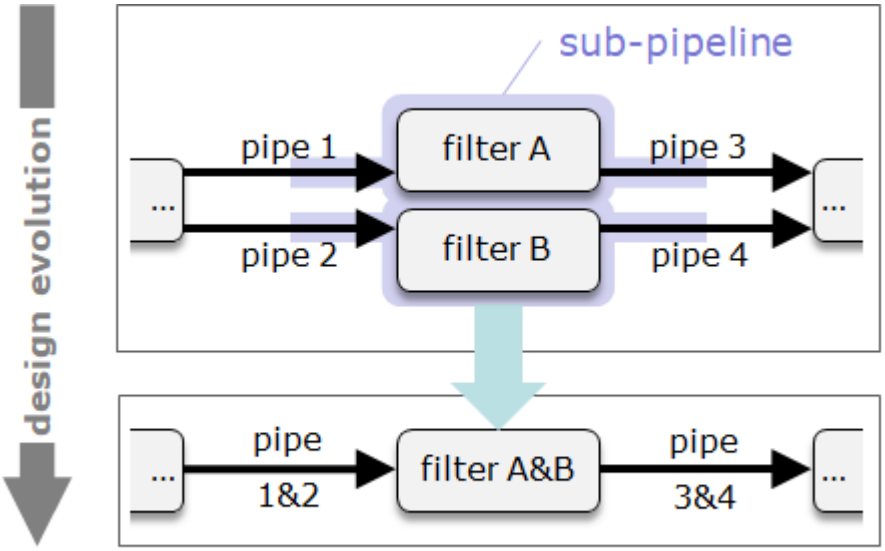


One can view the nesting in a diagram — as shown below.

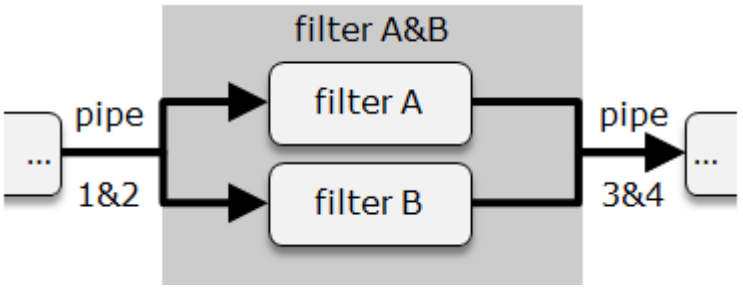


1.2.6.1 Nesting — pipe encapsulation

Where a nesting is going to encapsulate a number of filters, one consequence may be that a filter outside the nesting, whose pipes previously fed into multiple filters, now feeds into (or is fed from) a single encapsulated filter. In this case, the rule 'Multiple pipes between the same Filters' mentioned above comes into play, and the pipes need to be encapsulated to ensure in the nested pipeline there are not multiple pipes between the same (nested) filters — as shown in the figure below (as an aside, note the sub-pipeline is technically disconnected).



We can see the original multiple pipes in the nesting diagram — as shown in the figure below.



1.3 Nested gated pipeline architecture

1.3.1 Introduction

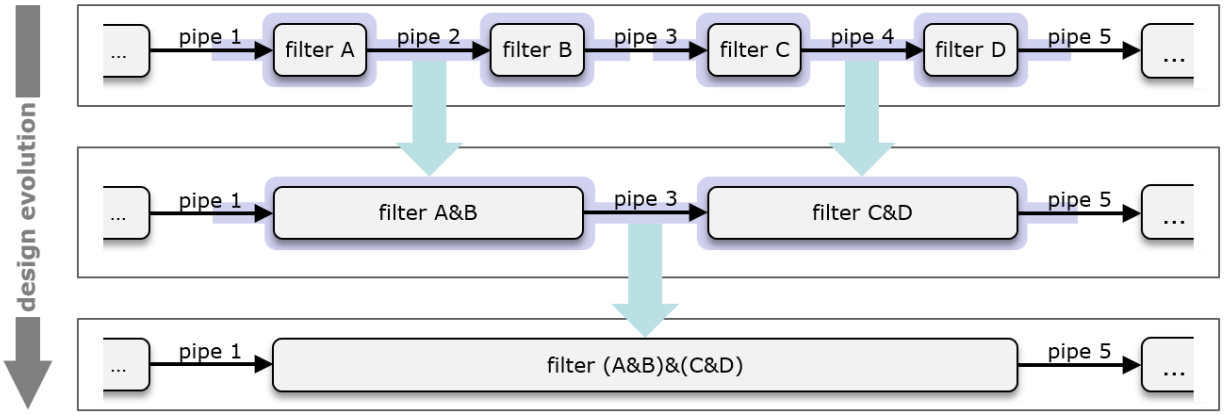
bCLEARer pipelines have a hierarchical, nesting structure. Within this pipeline architecture, the bCLEARer pipeline has a gated nesting structure. In the first part of this section, we look at how the pipelines nest. In the second part, we look at the gating strategies. The appendix subsection - Gate Object Accounting (see page 112) - explains the use of this architecture in object accounting.

1.3.2 The bCLEARer pipeline's multi-level nesting structure

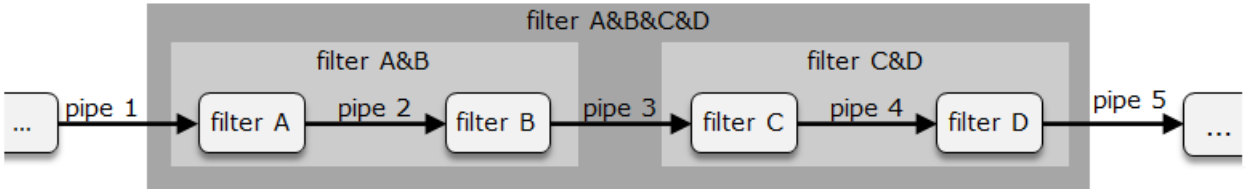
How the bCLEARer pipeline nests is described in this section.

1.3.2.1 Multi-level nesting

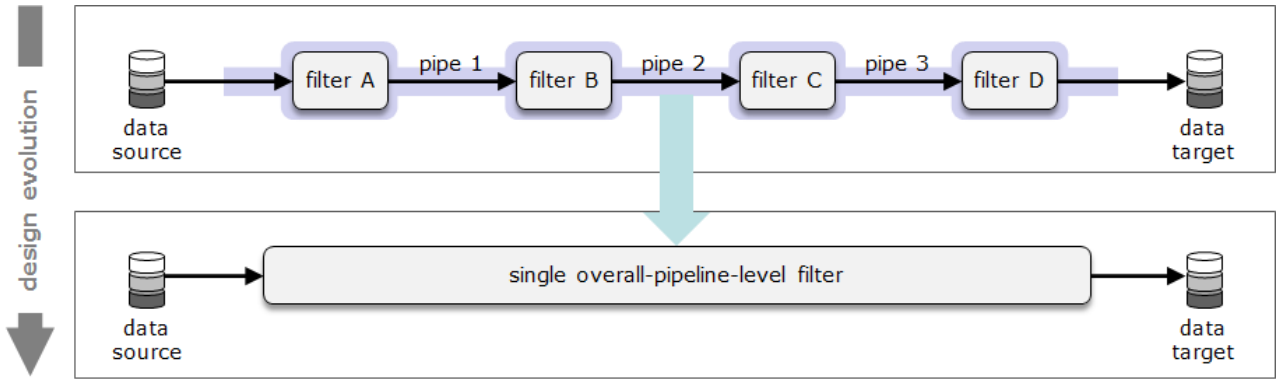
As noted earlier, within a pipeline architecture, one can encapsulate sub-pipelines as filters (with pipes). These encapsulated components can themselves be nested, creating a multi-level nesting (breakdown) structure in the pipeline.



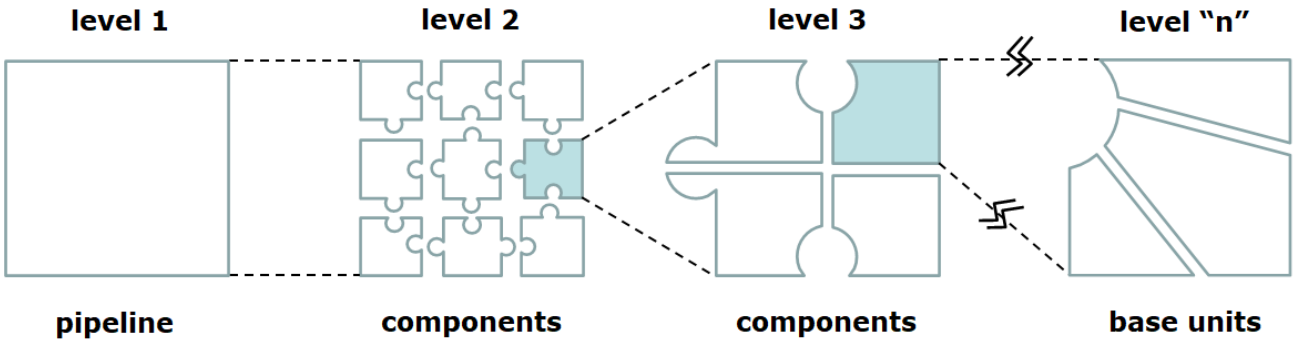
One can view the encapsulation structure in a nesting diagram — as shown below.



The final nesting is of all the filters into a single overall-pipeline-level filter.



From the perspective of this top level final pipeline, the nesting is a series of decompositions. These can be a fixed series of decompositions into levels, where each decomposition takes the units at one level and divides them into units at the next level. This breaks down the overall process into levels, with base (undivided) units at the bottom level — as shown in the figure below.



levelled breakdown

1.3.2.2 bCLEARer core nesting architectural levels

As discussed in the next section, Architectural nesting breakdown (see page 21), the bCLEARer pipeline's nesting has an architecture. It is initially composed of three core levels based on the type of filter:

levels	descriptions
pipeline	the top, root, level of nesting for the bCLEARer pipeline - there is only a single root pipeline filter at this level. It is the upper boundary of the pipeline architecture.
stage	a single boundary layer (with no nesting) that contains the filters for the stages of the bCLEARer digital journey.
bUnit	the leaf nodes in the nesting structure, the bUnit filters. As this is a hierarchical tree structure, they belong to one, and only one, bCLEARer stage. It is the lower boundary of the pipeline architecture.

(see page 225)

Between these levels the nesting can be extended to accommodate further modularisation.

1.3.3 Nested pipeline's data stage gates

The nested pipeline data stage gates are described in this section.

1.3.3.1 The data stage gates

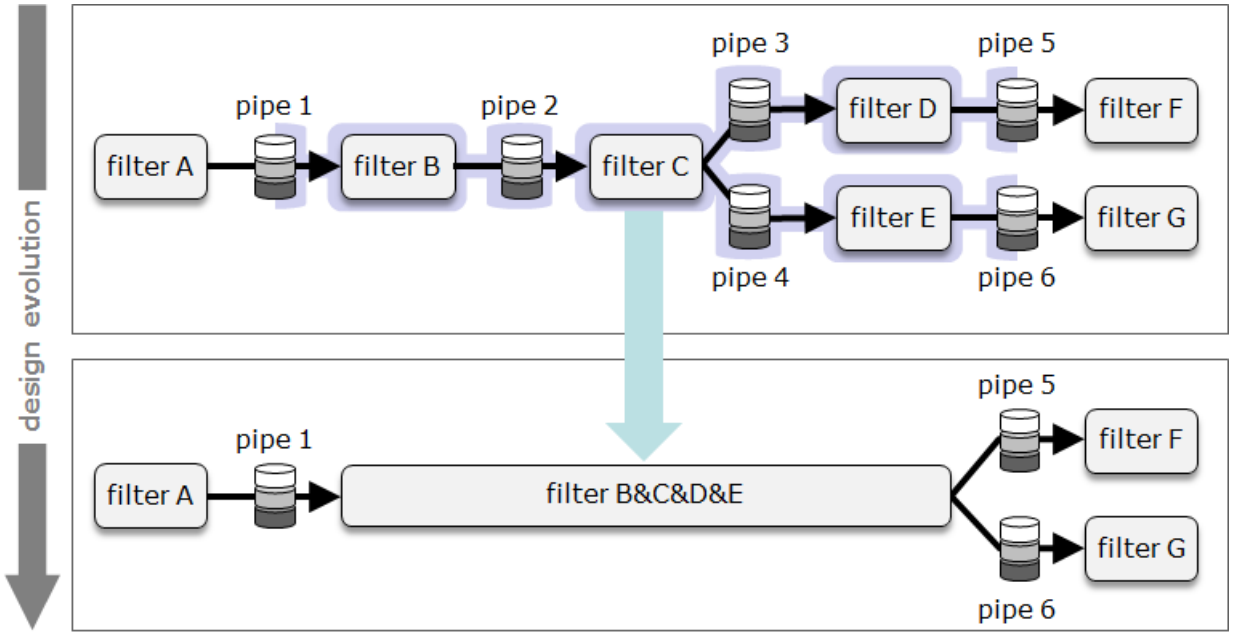
Where appropriate, the nested pipelines are designed with their input and output pipes as data stage gates. Typically, bCLEARer stage and thin slice pipelines have these gates. This dual (input and output) gate design for nested pipelines enables the original and transformed data to be inspected and compared. To assist with this, the design of an output gate's data also aims to clean it sufficiently to provide a SSOT (Appendix - Aggregated Single Source of Truth Principle (see page 46)) snapshot of its state at this stage of its digital journey. Where an input gate is an output gate of the previous process, it will also be in a SSOT snapshot. And in so doing, one can make the journey's transformations visible by comparing snapshots. In the bCLEARer process, these stage-gates are not decision points (as they are in some waterfall processes); though where data fails an inspection, it may be held back if required. So the focus is on inspection and not decision in an agile iterative process.

Establishing data gates for nested pipelines, especially SSOT data gates, is a very cost effective way of creating inspection points that can greatly simplify improving and maintaining quality. It helps, for example, in the identifying of the source of problems: finding the first gate at which a problem appears, isolates between which gates it arose. One of the drivers in the design of the nesting structure is building a hierarchy of levels with structured gaps between gates that enable one to drill down into processes so finding problem data is easier.

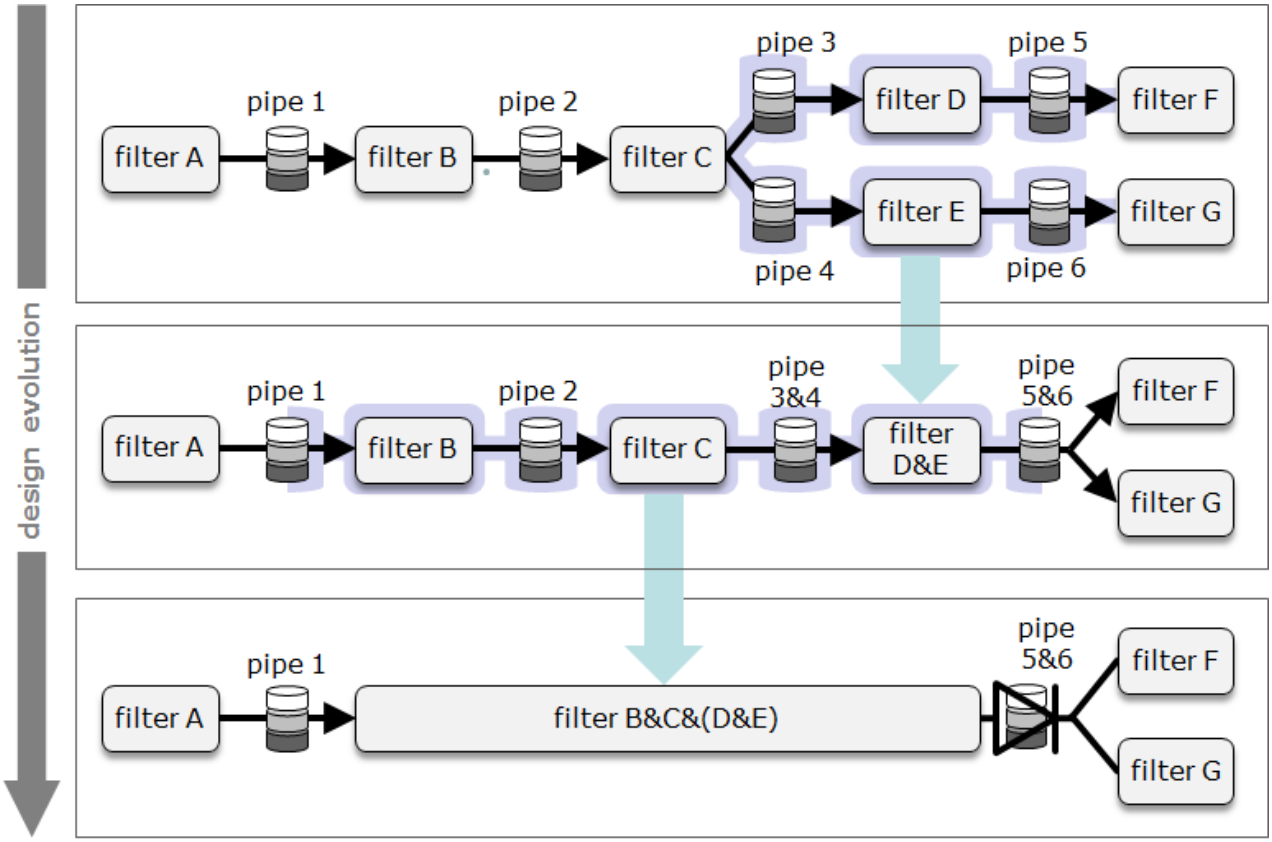
1.3.3.2 Designing a gate

Hence, gates are designed into the process. The design involves two layers of nesting. Given a pipeline that needs to be gated at one end, one needs to create a pipe that aggregates all the data flowing through that end of the pipeline. One way of doing this is creating a nesting that encapsulates all the filters with pipes that travel outside the pipeline. This will encapsulate the outgoing pipes into a single (aggregated) pipe, an example of this is given in the figures below.

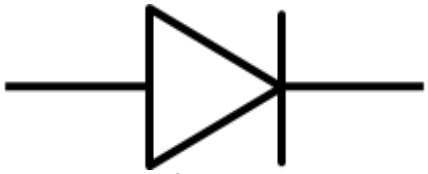
In this figure, the filters are just encapsulated. This results in the unaggregated output of two pipes — pipe 5 and pipe 6.



In this figure, the pipes are encapsulated first and then the filters. This results in the aggregated output of a single pipe — pipe 5&6.

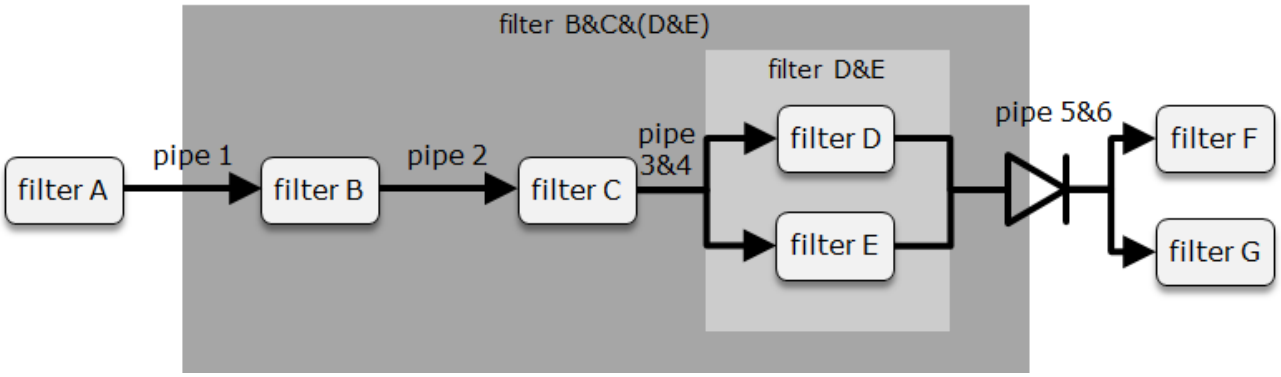


This diagram includes the gate icon, as shown below.



The data icon is typically adorned with a gate icon if the data is gated, as shown below.

As usual, we can look at a nesting diagram to see the lower level structure hidden in the simple pipeline diagram - as shown in the figure below.



1.4 Architectural nesting breakdown

1.4.1 Introduction

In the previous section, we noted that bCLEARer pipeline architecture has a hierarchical, levelled nesting structure. In this section, we look in more detail at this. The levels are an essential part of the stable bCLEARer infrastructure, the environment, within which the transformations evolve.

Sometimes it is conceptually cleaner to describe the hierarchy from the bottom up starting from the base unit. In practice, the design often starts with the overall scope and decomposes this into base units. We follow this design practice in this section, where we describe this nesting by level from the bCLEARer Pipeline down.

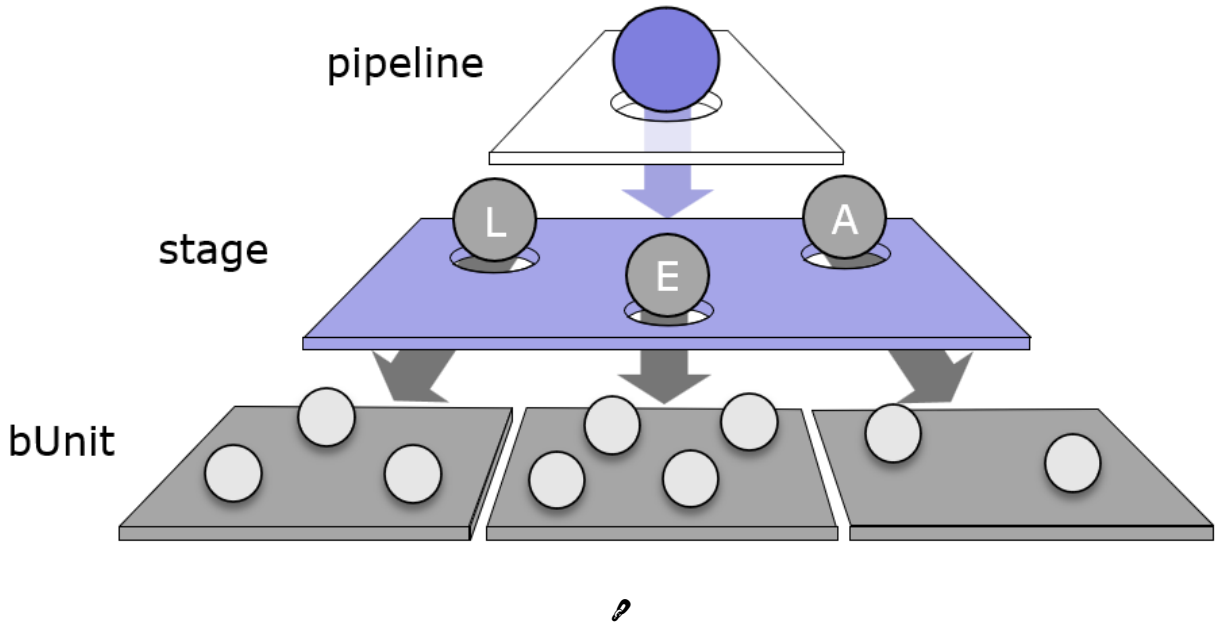
1.4.2 Three core levels

The bCLEARer pipeline's primary nesting is based upon three core levels - described in the table below.

levels	descriptions
pipeline	the top, root, level of nesting for the bCLEARer pipeline - there is only a single root pipeline filter at this level. It is the upper boundary of the pipeline architecture.
stage	a single boundary layer (with no nesting) that contains the filters for the stages of the bCLEARer digital journey.
bUnit	the leaf nodes in the nesting structure, the bUnit filters. As this is a hierarchical tree structure, they belong to one, and only one, bCLEARer stage. It is the lower boundary of the pipeline architecture.

(see page 225)

This can be visualised as a three-level breakdown.



1.4.3 Two extended levels

The pipeline can be extended between these core levels with two further secondary nesting levels - described in the table below.

levels	descriptions
thin slice	where there are a significant number of sequences of bCLEARer stages in a pipeline, inserting this level between the pipeline and the stages enables the stages to be grouped into a more hierarchical modular structure.
sub-stage	where a bCLEARer stage contains a significant number of bUnits, inserting this level between the stages and the bUNits enables the bUnits to be grouped into a more hierarchical modular structure.

(see page 226)

1.4.4 Level type faceting

The types of level in the pipeline nesting architecture can be classified under the three facets described in the table below.

Facet	Elements	Description
kind	core	mandatory core levels in the architecture that set the framework
	extension	optional extension levels that enable modularisation in the architecture
modality	mandatory	a mandatory level in the pipeline nesting architecture
	optional	an optional level in the pipeline nesting architecture
nestability	boundary slice	a single level - with no nesting - in the pipeline nesting architecture
	nestable	nesting is allowed - but not mandatory - in the pipeline nesting architecture

(see page 224)

The types of level are classified under these facets as shown in the table below.

level types	kind	modality	nestability
pipelines	core	mandatory	boundary (single)
thin slices	extension	optional	nestable
stages	core	mandatory	boundary (single)
sub-stages	extension	optional	nestable
bUnits	core	mandatory	boundary (single)

(see page 223)

1.4.5 Core levels

The bCLEARer pipeline's primary nesting is based upon the three core levels described in this section. The next section describes the two extended levels.

- Pipelines level (see page 25)
- Stages level (see page 26)
- bUnits level (see page 27)

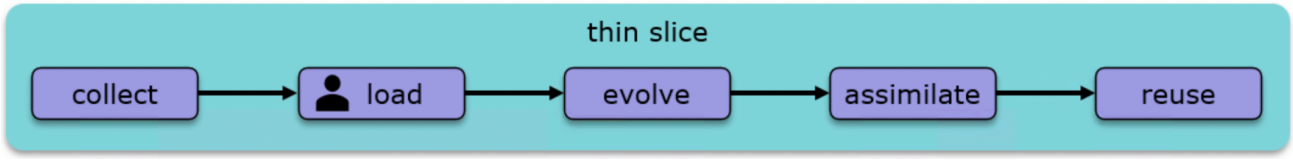
1.4.5.1 Pipelines level

The pipelines level is the top, root, filter of the bCLEARer pipeline. It is the upper boundary of the pipeline architecture. Every bCLEARer pipeline has this root filter.

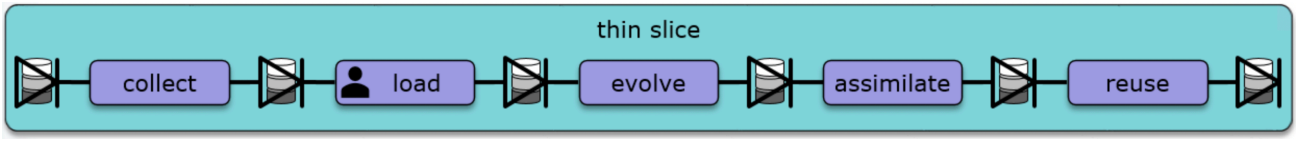
1.4.5.2 Stages level

The (bCLEARer) stages are a single boundary layer (with no nesting) that contains the filters for the stages of the bCLEARer digital journey.

bCLEARer provides a clear structure of five stages for the digital journey. These bCLEARer stages are sequential — and so the pipeline should flow in a sequence that reflects the journey. The sequence is well established (see first figure below).



The bCLEARer stages are usually gated — as shown in the figure below.



Sometime the pipeline involves manual work. In bCLEARer stage diagrams we usually mark the stages that involve manual work, using an icon — this is visible in the Load stage in the diagram above. The manual icon is shown below.

manual icon

As manual work interferes with running, scaling and costs, the aim is, as far as possible, to automate this work. Where it cannot be automated, it is often a good idea to move the manual work to the early stages in the sequence, typically restrict it to the Load stage.

1.4.5.3 bUnits level

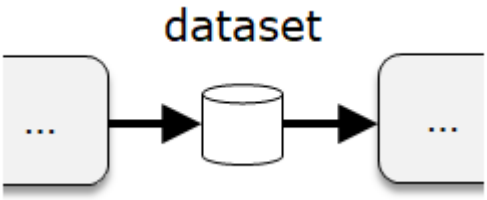
The bUnit filters are the leaf nodes in the nesting structure. Together, they form the lower boundary of the pipeline architecture. As this is a hierarchical tree structure, they belong to one, and only one, bCLEARer stage.

1.4.5.3.1 Base unit of transformation

In the bUnit pipeline architecture, the bUnit pipes are the base units of identity and difference, and the bUnit filters the base units of transformation. In this architecture, the pipes transport data, the data is not transformed — so it is immutable stage of the data through the flow. The flow can then be seen as a sequence of bUnit immutable stages, where any transformation is located in the filters linking the stages.

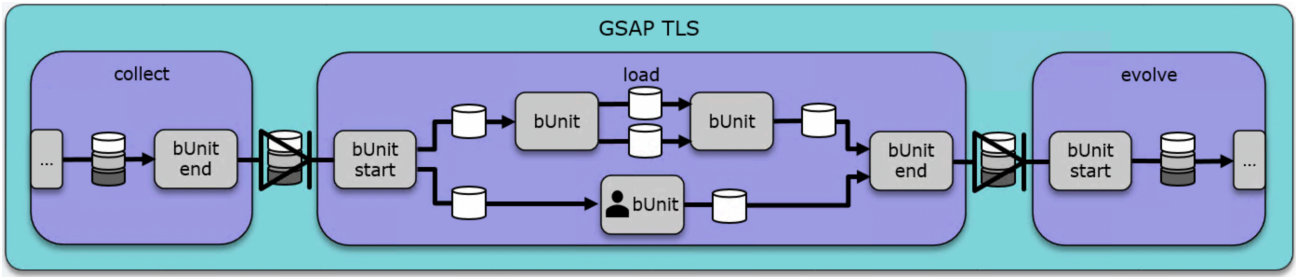
1.4.5.3.1.1 Finer-grained level of datasets

The pipes in the base bUnits work at the finer-grained level of datasets rather than dataset collections. In diagrams, the bUnit pipes are adorned with the dataset icon, as shown below.



1.4.5.3.2 Stage gates

Where the bCLEARer stages are gated (as they usually are), the bUnits need to be designed to accommodate the gates — this is shown as start and end bUnits in the figure below, as are the finer-grained datasets.



Sometimes, unavoidably, a bUnit will be manual. This is marked using the manual icon — there is an example of this in the diagram above.

1.4.6 Extended levels

The pipeline can be extended between these core levels with two further extended levels, described in this section.

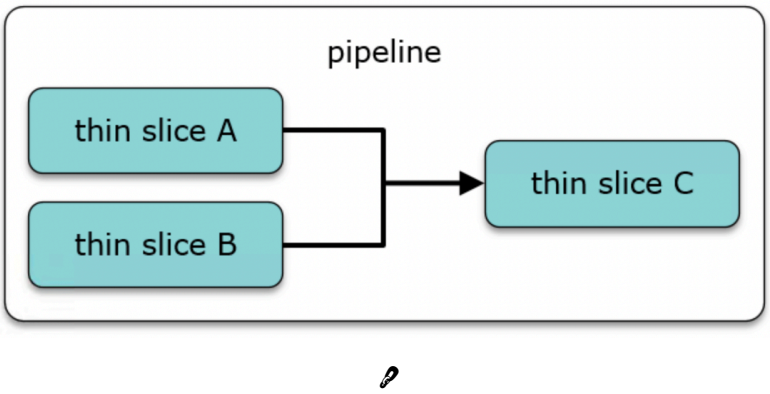
- Thin slices level (see page 29)
- Sub-stages level (see page 33)

1.4.6.1 Thin slices level

Where the breakdown from the pipeline to the (bCLEARer) stages involves a significant number of stages, then it makes sense to modularise these stages into a nesting hierarchy of sub-pipeline filters. This is done in the thin slice extended level and built by nesting the stages in a hierarchy of thin slices.

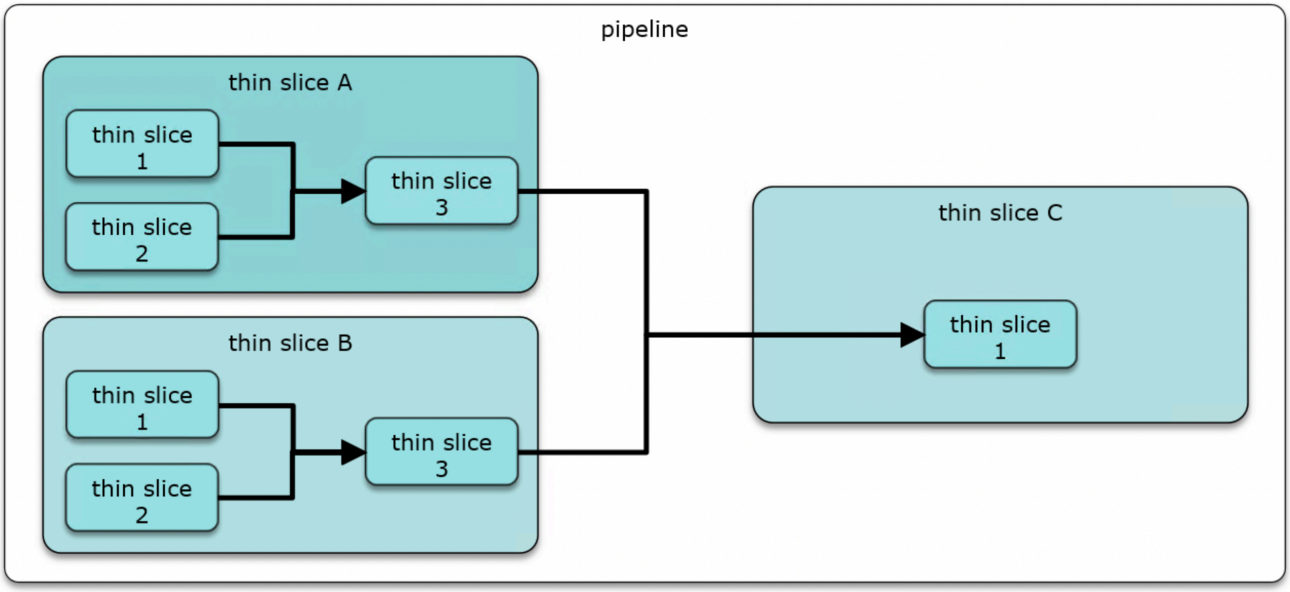
1.4.6.1.1 Thin slice pipeline decomposition

When designing the nesting structure for the thin slice level, it often makes sense to start with the overall bCLEARer pipeline and make a series of decompositions down until one reaches its boundary level of stages. A pro-forma example of a first stage decomposition of a bCLEARer pipeline into thin slices (filters), where the flow is ordered by pipes is shown in the figure below.



1.4.6.1.1.1 Thin slice nesting

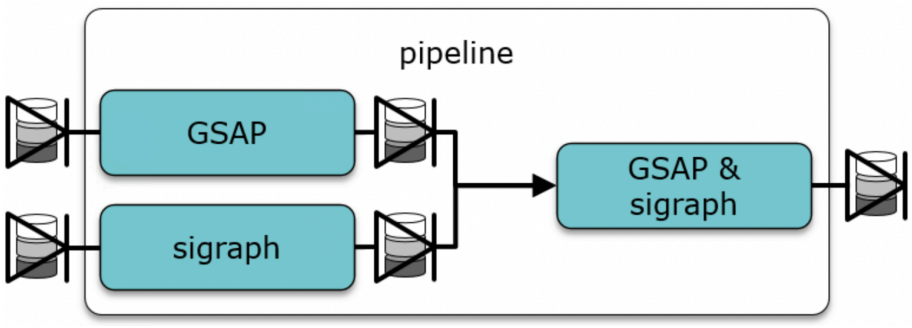
Where needed a thin slice can be further broken down, creating multiple layers of thin slice nesting. The breakdowns are ordered using pipes, reflecting the dependencies between the chunks - as shown in the pro-forma nesting diagram in the figure below.



1.4.6.1.2 Domain-based thin slicing

It is not uncommon for the bCLEARer pipeline to cover more than one domain and in these cases the thin slice decompositions are often driven by domain considerations. For example, it is a common practice when working with multiple domains, to start bCLEARing each domain independently before merging them, especially so if the domains are sizeable.

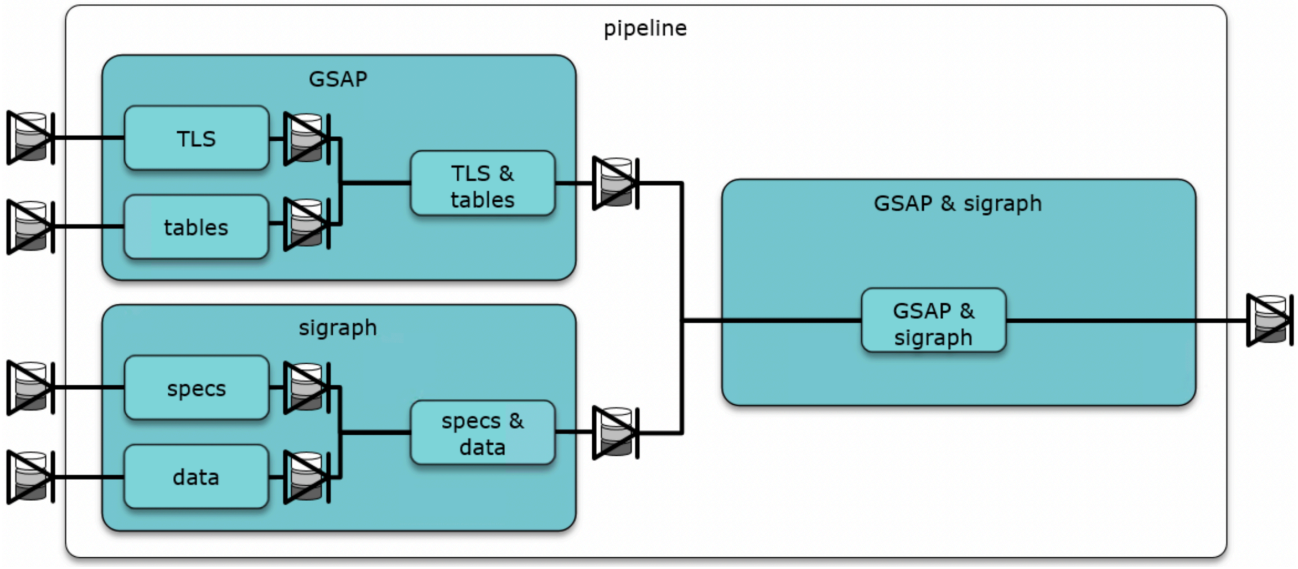
We illustrate this in the figure below, which involves two systems: GSAP and Sigraph. The first two thin slices transforms the two systems in isolation. The third thin slice merges and transforms the data from the two initial thin slices. Systems and their sub-systems often form natural boundaries for thin slices. Gates are typically placed at the start and end of each thin slice – as shown in the figure below using the dataset collection icon.



1.4.6.1.3 Intra-domain thin slicing

With any sizeable domain, it often makes sense to divide it into sub-pipelines based upon smaller intra-domain chunks. Where there are dependencies between the chunks, these need to be recognised in the bCLEARer pipeline flow.

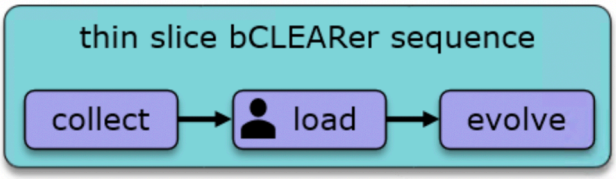
We illustrate this in the figure below, which again involves two systems: GSAP and Sigraph. This time we consider chunks within the system domains. Again, gates are typically placed at the start and end of each thin slice.



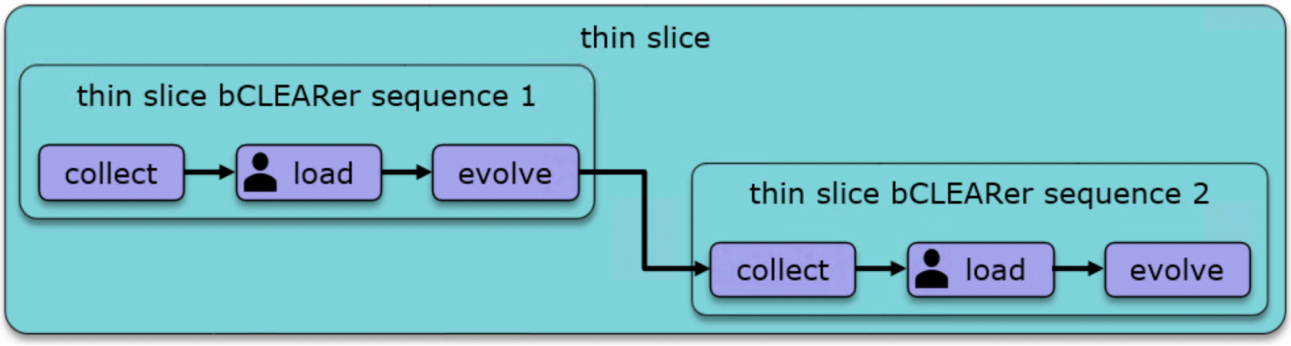
This figure shows the merging of data within systems as well as across systems. This is not unusual, often a significant amount of the bCLEARer pipeline deals with the merging of data from different sources - sometimes relating to the same domain or systems, other times across different domains or systems.

1.4.6.1.4 Thin slice bCLEARer sequences

The leaves of a thin slice nesting hierarchy are bCLEARer stage (Stages level (see page 26)) filters. Within the thin slice, these are typically ordered by pipes into a bCLEARer sequence - as shown in the example below.

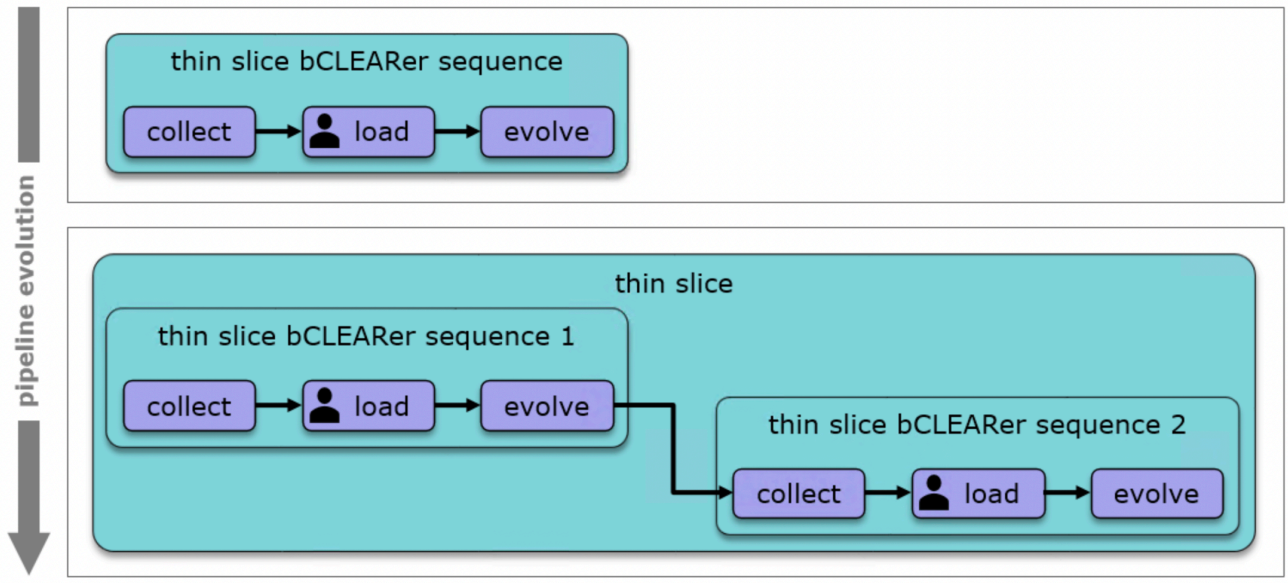


Where there are multiple thin slices these are ordered by pipes into a series of sequences, where one sequence follows another — see example below.



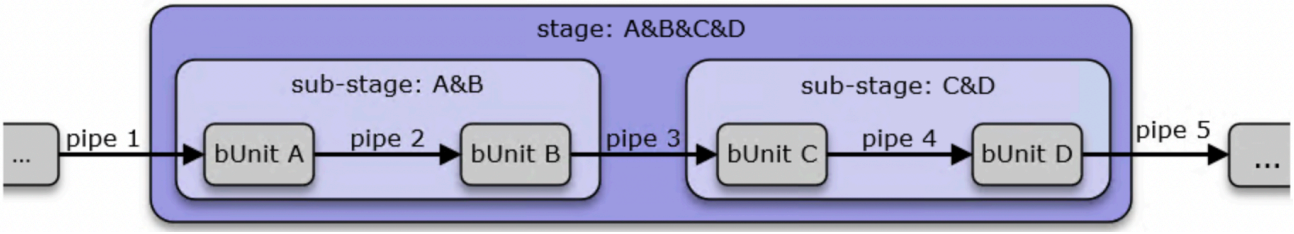
1.4.6.1.5 Thin slice hierarchy evolution

Also, the thin slice pipeline will typically evolve during a project — as shown in the example below.



1.4.6.2 Sub-stages level

Where a bCLEARer stage contains a significant number of bUnits, the sub-stages level is introduced to group the filters into a more hierarchical modular structure.



1.5 Core design practices and patterns

1.5.1 Introduction

This section deals with the general design of the bCLEARer pipeline to facilitate transparency and resilience. The bCLEARer pipeline is designed as a sequence of nested filters - the pipeline flow. It is important that this flow is not only transparent, open to inspection, but also resilient in the face of change. Hence we discuss the general design principles that will encourage this.

1.5.2 Design principles

We have found it useful to design the bCLEARer pipeline with a view to reducing the cost of evolving the pipeline - especially radically evolving - to take advantage of the opportunities that emerge on the journey of discovery. There are many well-known principles and techniques to achieve this. For example, focusing on building in modularity not only increases maintainability and reusability, but also improves refactoring and extensibility. These are well-documented, and we aim to point to these resources here and in the associated appendices.

However, our focus here is on the less well-documented goal of facilitating transparency. One core central motivation for this is that delivering transparent transformations, and so enabling easy inspection, is a solid basis for improving the value of, as well as enhancing the trust in, the transformations. The aim is to systematise this inspection so reducing the effort needed to develop and maintain it during the rapid evolution of the pipeline.

In the following sections, we aim to highlight here with some examples ways we can use the many well-known principles to better achieve our more focused goals. We start with a look at the general approach to decomposition, and then look at two principles:

- separation of transformation concerns
- immutability and idempotence

There is a significant literature on these principles, so here we outline them and give references for further reading in the associated appendices.

1.5.3 General approach to decomposition

In previous sections, we have visualised the modular structure of pipeline flow in terms of a flowchart to provide a picture of its structure. However, we need a different approach to decomposing it into filters. To reinforce this point, consider David L. Parnas's classic Communications of the ACM paper (Parnas 1972 Decomposing Systems into Modules) (see page 139) where he makes this point, proposing an alternative to simple flowchart decomposition saying:

> "We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

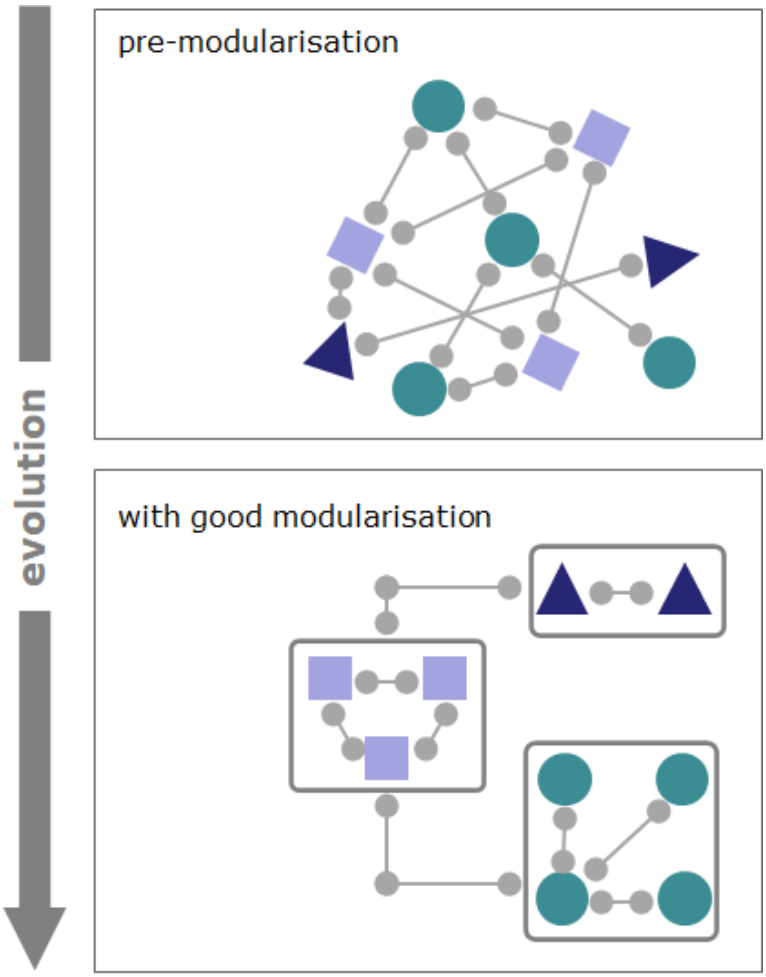
In the case of bCLEARer, the design decisions should consider, at the least, the various different kinds of information transformation. At the lowest level, the breakdown should end up with bUnits that deal with a single transformation - bUnits should not handle more than one transformation. This could be regarded as a "single-transformation principle" - alluding to the single-responsibility principle. For more details see: Appendix - Single-Transformation (Responsibility) Principle (STP) (see page 78).

1.5.4 Principle: separation of transformation concerns

The separation of concerns (described in more detail in the Appendix (Appendix - Separation of Concerns Principle (see page 75))) is similar in many respects to single-transformation principle (see: Appendix - Single-Transformation (Responsibility) Principle (STP) (see page 78))). From a concerns perspective, the different types of transformation can be seen as different concerns, and so should be separated. As a first stage, this leads to base bUnits with a single concern. This enables cleaner tracing of transformations, as it also separates out the contributors to the transformation.

This can lead to a significant number of base bUnits. When organising these base bUnits into larger modules, it is useful to consider principles of cohesion and coupling (there are more details on this in Appendix - Loose Coupling and Tight Cohesion Principle (see page 54)).

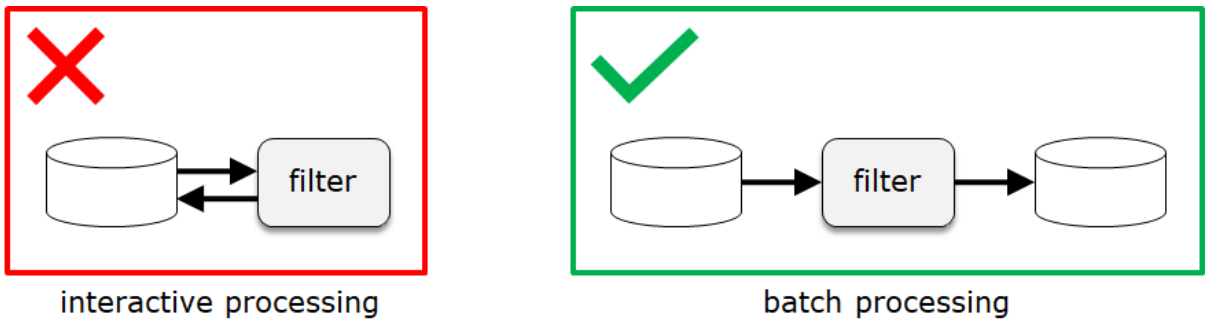
Cohesion is a measure of functional closeness - and a good modular design will have high cohesion where the functional closeness is high inside the modules and low outside. Coupling is a measure of interdependence. A good modular design will have low coupling between modules and higher coupling inside the module. This is illustrated visually in the figure below.



1.5.5 Principle: immutability and idempotence

Immutability and idempotence are about how transformations are handled. Immutable data's content cannot be modified after it is created. An idempotent process can be run multiple times with the same input without changing the output. The pipeline architecture promotes idempotent processes (filters) and immutable datasets.

We can illustrate this by comparing interactive and batch processing. The data updated by interactive processing is typically not immutable - the process changes its content. Whereas the data created by batch processing - the common pipeline process - is typically immutable. This is shown in the graphic below.



interactive processing batch processing

So, adopting immutable data and idempotent processes (filters) is a natural choice for a pipeline architecture. And adopting it has several advantages. The immutability simplifies auditability, making transformations easier to track. The idempotence makes rerunning the processes safer and simpler. For more on these topics, see the appendix: Appendix - Immutability and Idempotence Principles (see page 50).

1.6 Appendices

1.6.1 Appendix - The Standard 'Pipeline' or 'Pipe-and-Filter' Architecture

'Pipe-and-filter' is a software architecture pattern. The pattern is used to break down a complex monolithic process into individual and reusable components. Doing so can improve performance, scalability, and reusability by allowing task elements that perform the processing to be deployed and scaled independently. The name "pipeline" comes from a rough analogy with physical plumbing in that a pipeline usually allows information to flow in only one direction, like water often flows in a pipe.

This appendix discusses the standard 'Pipeline' or 'Pipe-and-Filter' Architecture in the following three sections:

1.6.1.1 The standard 'Pipeline' or 'Pipe-and-Filter' Architecture - History

Doug McIlroy introduced the ‘pipe-and-filter’ architecture in Unix in 1972. (Ritchie 1984 The Evolution of the Unix Time-Sharing System) (see page 150) gives a detailed description of this. It subsequently became known as ‘pipeline’ architecture. Since then it has become an accepted architecture or architectural pattern and documented in the standard textbooks; an example is given at the end of the section.

1.6.1.1.1 Compliers

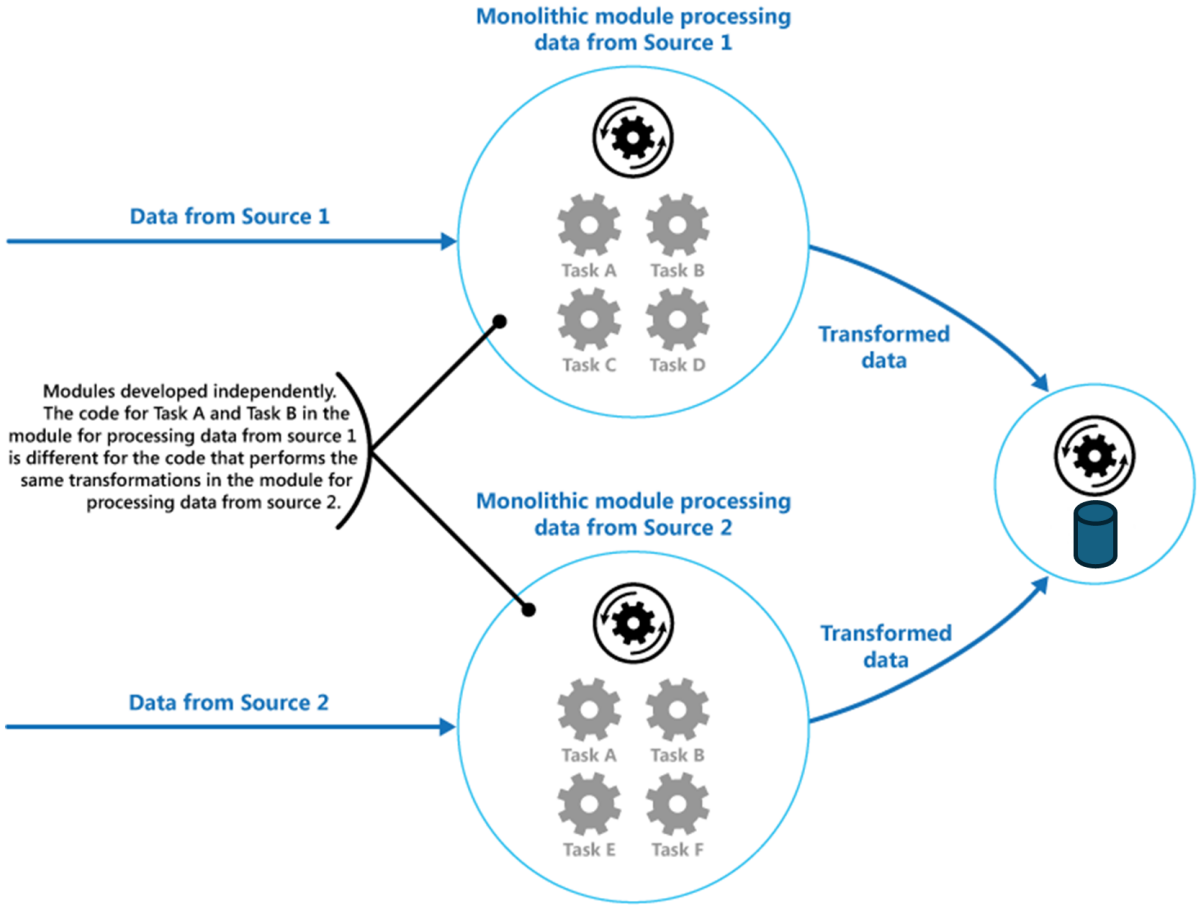
Particular applications often have a more detailed architecture. For example, compilers typically have a three stage pipeline architecture:

1. front-end: parses input language into an intermediate language
2. middle: performs transformations in the intermediate language
3. back-end: translates the intermediate language into the output language

The use of an intermediate language allows for reusable architectural components.

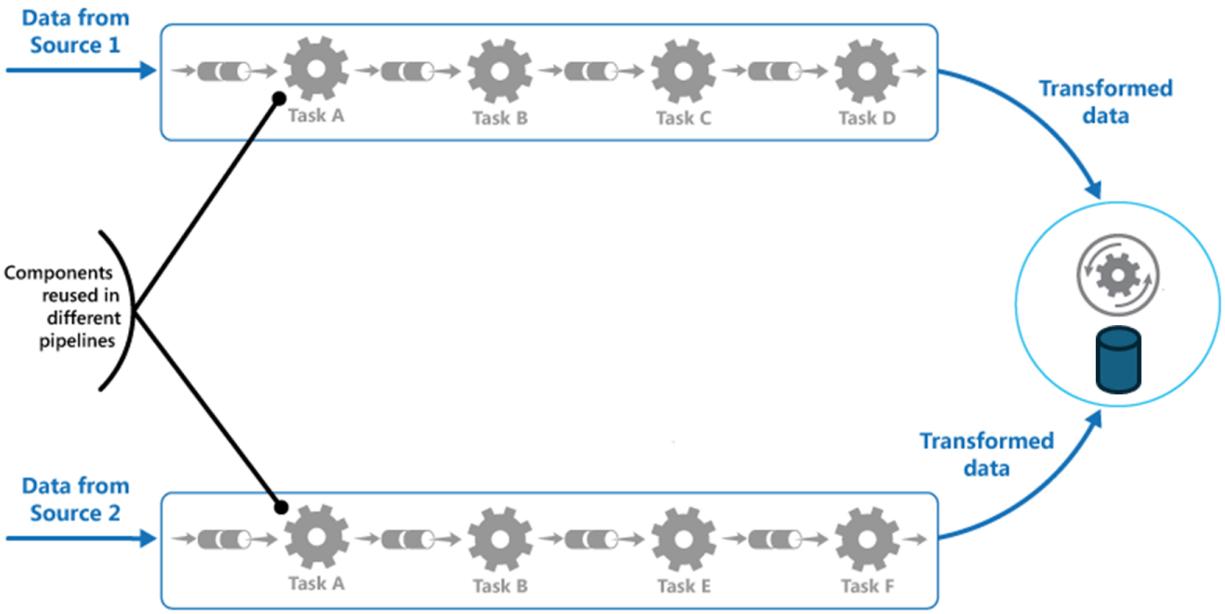
1.6.1.2 The standard 'Pipeline' or 'Pipe-and-Filter' Architecture - Pipe-and-filters breakdown approach

A straightforward approach to implementing an application is to perform each processing task in a module. This results in an inflexible, monolithic architecture, which is likely to reduce the opportunities for reusing the code and so enhancing the value of refactoring and optimizing it. The diagram below illustrates this monolithic approach.



The two modules were designed separately and the code that is closely coupled to its module.

A pipe-and filter architecture breaks down the processing for each stream into a set of separate *filters*, each performing a single task. The filters are then combined into a pipeline. This not only avoids code duplication, but makes it easy to remove or replace or integrate additional filters, if the requirements change. This diagram below shows a solution that's implemented with pipes and filters:



1.6.1.2.1 Recognised benefits

As the diagrams above help to illustrate, there are a number of recognised benefits of using this pattern:

- Ensures loose and flexible coupling of pipes and filters.
- Loose coupling allows filters to be changed without modifications to other filters.
- Conductive to parallel processing.
- Filters can be treated as black boxes. Users of the system don't need to know the logic behind the working of each filter.
- Re-usability. Each filter can be called and used over and over again.

1.6.1.3 The standard 'Pipeline' or 'Pipe-and-Filter' Architecture - Standard Textbook Example

(Bass 2012 Software Architecture in Practice) (see page 121) is one of the standard textbooks on software architecture. It describes architectures in terms of patterns. This extract provides the 'standard' view of pipe-and-filter architecture.

The standard pipe and filter pattern

Pipe-and-Filter Pattern

Context: Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.

Problem: Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.

Solution: The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

> *Data transformation systems are typically structured as pipes and filters, with each filter responsible for one part of the overall transformation of the input data. The independent processing at each step supports reuse, parallelization, and simplified reasoning about overall behaviour. Often such systems constitute the front end of signal-processing applications. These systems receive sensor data at a set of initial filters; each of these filters compresses the data and performs initial processing (such as smoothing). Downstream filters reduce the data further and do synthesis across data derived from different sensors. The final filter typically passes its data to an application, for example providing input to modelling or visualization tools*

Key terms

terms	descriptions
Overview	Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.

terms	descriptions
Elements	*Filter*, which is a component that transforms data read on its input port(s) to data written on its output port(s). Filters can execute concurrently with each other. Filters can incrementally transform data; that is, they can start producing output as soon as they start processing input. Important characteristics include processing rates, input/output data formats, and the transformation executed by the filter. *Pipe*, which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through. Important characteristics include buffer size, protocol of interaction, transmission speed, and format of the data that passes through a pipe.
Relations	The *attachment* relation associates the output of filters with the input of pipes and vice versa.
Constraints	Pipes connect filter output ports to filter input ports. Connected filters must agree on the type of data being passed along the connecting pipe. Specializations of the pattern may restrict the association of components to an acyclic graph or a linear sequence, sometimes called a pipeline. Other specializations may prescribe that components have certain named ports, such as the *stdln*, *stdout*, and *stderr* ports of UNIX filters.
Weaknesses	The pipe-and-filter pattern is typically not a good choice for an interactive system. Having large numbers of independent filters can add substantial amounts of computational overhead. Pipe-and-filter systems may not be appropriate for long-running computations.

(see page 123)

1.6.2 Appendix - Aggregated Single Source of Truth Principle

The *single source of truth* (SSOT) principle is also called *single point of truth* (SPOT) principle.

1.6.2.1 Two versions of the principle

There are syntactic and semantic (or ontological) versions of this principle. In the syntactic version, this is the practice of structuring information models and associated data schemas such that every data element is mastered in only one place. In the semantic (or ontological) version, there is a further assumption that objects in the real world are only represented once by a single data element.

1.6.2.2 Aggregated single source of truth

The need for an aggregated SSOT arises when there are multiple systems. While each of the systems may have implemented a SSOT policy, new requirements emerge when the information in the systems are aggregated. From a syntactic perspective, one can often identify when data elements in different systems are (in some sense) the same - and set up processes to aggregate them.

From a semantic (ontological) perspective, there is a requirement to aggregate (syntactically) different data elements that represent the same object in the real world. Identifying where the same real world object is being represented is often greatly simplified by adopting a top ontology.

1.6.2.3 BORO and bCLEARer's approach

bCLEARer projects often work with multiple systems - in fact, this is encouraged as the use of multiple systems increases the quality of the resulting ontology. bCLEARer's integrated use of the BORO Top Ontology greatly simplifies semantic aggregated SSOT. Its integrated use of object-identity enables transparent inspection of the aggregation process.

(see page 46)

1.6.3 Appendix - Glossary of Major Terms

1.6.3.1 Pipeline architecture

terms	descriptions
bCLEARer pipeline	A nested pipeline that has thin slices, bCLEARer stages and bUnits as its decomposition (nesting) levels.
filter	See pipe-and-filter.
gate (gated pipeline)	Also known as data stage gate or information gate, this is a pipe that has been designed to enable all the original and transformed data to be inspected and compared.
nest (nested pipeline)	A pipeline with collapsed connected sub-pipelines.
pipe	See pipe-and-filter.
pipe-and-filter	Pipe-and-filter is a sequence of processing components (filters), arranged so that the output of each component is the input of the next one, connected by pipes, creating a ‘flow’. This is also known as a pipeline.
pipeline	See pipe-and-filter.
sub-pipeline	Any subset of filters in a pipeline.
pipeline topology	The network structure of the relations between pipes and filters in the pipeline

1.6.3.2 Nested levels

terms	descriptions
3 core level architecture	The bCLEARer pipeline’s 3 core level architecture
pipeline level	The top level of the 3 core level architecture. A single, root filter.

terms	descriptions
(bCLEARer) stage level	The middle level of the 3 level architecture.
bUnit level	The bottom level of the 3 level architecture.
thin slice level	One of the two extended levels to the 3 core level architecture.
sub-stage level	One of the two extended levels to the 3 core level architecture.
bCLEARer stage	The base units (filters) of the bCLEARer stage level of the bCLEARer Pipeline. Each unit is one of bCLEARer's five stages of evolution. In sequence these are: Collect, Load, Evolve, Assimilate and Reuse.
bCLEARer (stage) sequence	The composite units (filters) of the bCLEARer stage level that contains some or all of the bCLEARer stage filters in sequence (see bCLEARer stage).
bUnit	The base units (filters) of the bUnits level of the nested bCLEARer Pipeline, and so the lowest, bottom units of the pipeline.
thin slice	Composite unit (filters) that contains either other thin slices or stages.
sub-stage	Composite unit (filters) that contains either other sub-stages or bUnits.

(see page 47)

1.6.4 Appendix - Design Patterns and Anti-patterns

As Wikipedia (https://en.wikipedia.org/wiki/Design_pattern) notes, a design pattern 'is the re-usable form of a solution to a design problem'. Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. The range of situations in which a pattern can be used is called its context.

In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

1.6.4.1 Anti-pattern

An anti-pattern in software engineering, project management, and business processes is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive.

As opposed to a bad practice

- An anti-pattern is a commonly-used process, structure or pattern of action that, despite initially appearing to be an appropriate and effective response to a problem, has more bad consequences than good ones.
- Another solution exists to the problem the anti-pattern is attempting to address. This solution is documented, repeatable, and proven to be effective where the anti-pattern is not.

(see page 49)

1.6.5 Appendix - Immutability and Idempotence Principles

Immutability and idempotence are two inter-related software principles. They both deal with how data transformations are handled: immutability is concerned with the data content and idempotence is concerned with its processing.

Both the immutability and idempotence principles play significant roles in software engineering, providing guidance on designing more predictable, fault-tolerant, and maintainable systems. It is recognised that together they contribute to creating robust architectures, especially in environments where consistency, reliability, and performance are paramount.

This appendix has three subsections, linked below. The first discusses immutability and the second idempotence. The final section explains how these principles are used within bCLEARer (and BORO) projects:

1.6.5.1 Immutability principle

The immutability principle applies to objects, typically in object-oriented (OO) and functional programming, but can easily be extended to ‘database objects’. The principle is that the object’s state is immutable - it does not change.

An example of immutability in computation would be a constant - such as the value of pi. Constants do not change during their lifetime in the pipeline.

One key benefit of immutability is simpler auditability. Having immutable data gives the content a clear identity making possible easy tracking through the processing it is involved in.

Immutability reduces temporal coupling which has various downstream benefits in software design. One of these benefits is that it enables multithreading by promoting thread safety - programs that have a lower coupling on when they need to execute with respect to each other may be more safely multi-threaded.

Usually we think immutable objects can be immutable simpliciter, globally immutable. However, there is a weaker notion of immutable, where objects are immutable with respect to a process. Repeated applications of this process to the immutable object must produce the same result. However, there may be other processes that change the state of the object. An object is globally immutable if there is no process that changes its state.

The notion of immutable needs a corresponding notion of (state) change and (object) creation. The creation (transaction) of an object may be an intricate affair, During the creation its attributes (its state) may change. The major attributes may be created as a first stage and the minor ones subsequently: this is not considered state change.

The extent of the immutability can be extended. This is easier to see with data objects. So not just a row (corresponding to an object) can be immutable, but the whole table can be immutable.

1.6.5.2 Idempotence principle

The idempotence in software engineering is a property of a process (operation), where the process (operation) can be applied multiple times without changing the result beyond the initial application. This principle is crucial for designing robust, fault-tolerant systems, particularly in distributed systems, RESTful APIs, and database transactions.

The principle of idempotence has a rich history in computer science and software engineering, with its roots in mathematics and logic. Understanding its evolution provides insight into its broad applicability and importance across different areas of technology.

The concept of idempotence originates from mathematics, specifically algebra, where an operation is idempotent if applying it multiple times has the same effect as applying it once. For example, multiplying a number by one or taking the absolute value of a value are idempotent operations. The term was introduced by the mathematician Benjamin Peirce in the context of elements of algebras that remain invariant when raised to a positive integer power, and literally means "(the quality of having) the same power", from (*idem + potence* = same + power).

In formal logic and set theory, the concept of idempotence has been applied to operations and functions. It was recognized that certain operations, when applied multiple times, do not change the outcome beyond the initial application. This concept was naturally extended into computer science, particularly in the development of algorithms and data processing techniques. As we discuss in the next section (Immutability and idempotence in bCLEARer (and BORO) (see page 53)), bCLEARer bUnits are analogous to idempotent operations and functions.

Idempotence has also influenced software design patterns and practices. It underpins many aspects of software development, including error handling, state management, and the implementation of idempotent functions and services. This ensures that systems are more resilient to failures and can recover more gracefully from errors.

Idempotence and immutability are related. Where an object is immutable with respect to a process, then the the process is idempotent with respect to the object. Repeated applications of the process to the immutable object must produce the same result. For instance, the program for calculating the area of a circle that treated pi as an (immutable) constant would not change the value of pi. The operation (calculating area) is idempotent (concerning pi).

1.6.5.3 Immutability and idempotence in bCLEARer (and BORO)

It is generally accepted that, following idempotence and immutability in system design reduces the costs associated with complex systems, leading to easier maintainability, easier development, less costly refactoring, easier auditability. It also reduces the risks associated with complex systems leading to safer processing of data. Understandably, given these benefits bCLEARer adopts these principles.

The resulting design of the bCLEARer pipeline has a “batch processing” structure, where data is input to a process and different data is created by the process - the input data is never updated. This ensures idempotence.

As noted in the section, Immutability principle (see page 51), one needs to be clear of the extent of the creation transaction. In the bCLEARer pipeline, the creation transactions occur in the base bUnits. After a dataset is output by the base unit, it is immutable. This is an example of immutability extended to tables (mentioned in the earlier section, Immutability principle (see page 51)).

1.6.6 Appendix - Loose Coupling and Tight Cohesion Principle

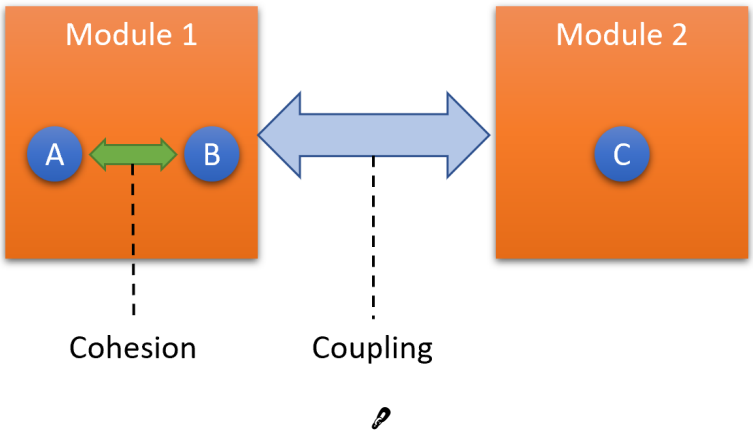
Loose Coupling and Tight Cohesion is one of the guiding principles of bCLEARer (see Appendix - Resilient Governance - The Right Balance of Principles and Rules (see page 69)).

1.6.6.1 History

Larry Constantine is considered to be the father of these concepts. It was first publicly described in his paper with Stevens and Myers: (Stevens 1974 Structured Design) (see page 152). It was originally described for procedural programming modules. It was later extended to cover object oriented programming classes.

1.6.6.2 What are cohesion and coupling?

The terms typically apply to the design of modules for software. Cohesion is a measure of how closely related parts of a module are functionally. One (rough) way of thinking of this is that a module with parts that work together for the same goal has high cohesion. Coupling is a measure of modules' (inter)dependance with one another. One (rough) way of thinking of this is that modules that are loosely coupled need each other less to function. This is shown graphically below.



This table summarises these points.

cohesion	coupling
Cohesion is typically an intra-module concept.	**Coupling is typically an inter-module concept.**
Cohesion is a measure for the relationships within a module.	**Coupling is a measure for the relationships between modules.**
Increasing (tightening) cohesion is good for software.	**Loosening coupling is good for software.**

1.6.6.3 Principle

There are close links between this principle and Appendix - Separation of Concerns Principle (see page 75).

1.6.6.3.1 Tight Cohesion as a principle

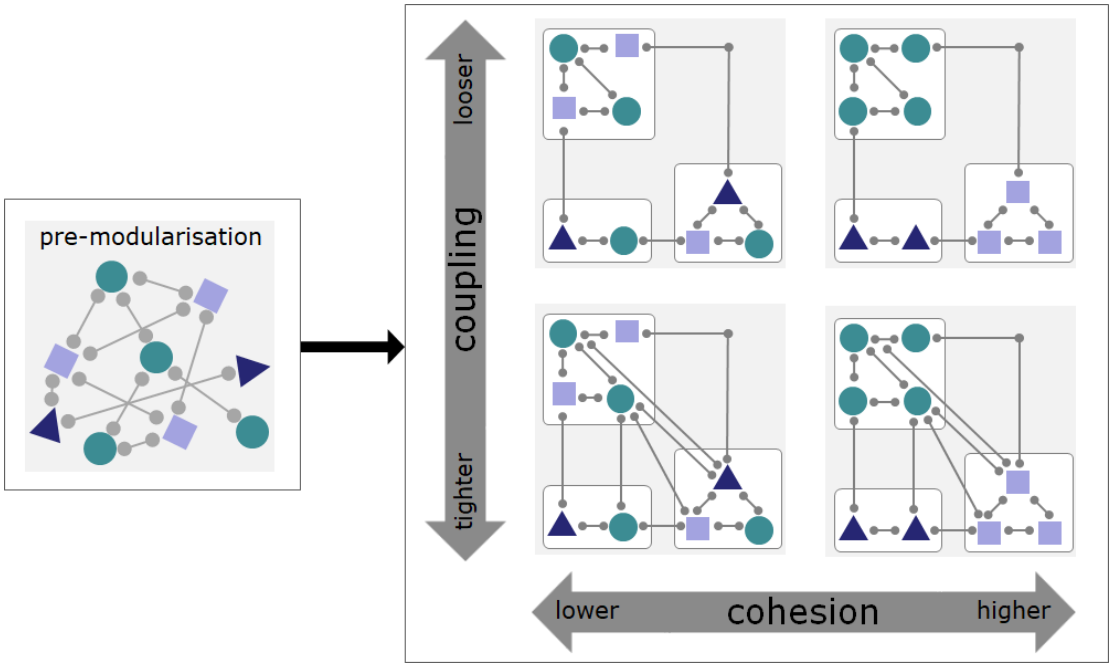
This requires that engineers design module parts so that they are functionally close. For instance, in OOP, making sure that a class is designed with a single, well-focused purpose.

> Suppose we have a class that multiplies two numbers, but the same class creates a pop-up window displaying the result. This is an example of a low cohesive class because the window and the multiplication operation don't have much in common.
>
> To make it tightly cohesive, we would have to refactor things. We would create a new class Display and Multiply, where Display creates a pop-up window displaying the result and Multiply multiplies two numbers. And then the Display class will call Multiply's method to get the result and display it. This is the way to develop a tight cohesive solution.

1.6.6.3.2 Loose Coupling as a principle

This requires that modules are designed so that they are less interdependent. For instance in OOP, this principle demands that each class should be responsible for its own operations in a process.

The figure below illustrates this graphically.



shifting to loose coupling and tight cohesion

1.6.6.3.3 Value of tight cohesion and loose coupling

This principle promotes modularity, reusability and scalability. It makes software easier to maintain and test.

1.6.6.4 Cohesion and Coupling in bCLEARer

bCLEARer extends this principle to pipeline design. At all levels, each unit should be tightly cohesive and the collections of units inside a pipeline should be loosely coupled - relative to each other. One way to do this is to base the unit groupings on 'concerns' - see Appendix - Separation of Concerns Principle (see page 75). A classic example of this is where multiple systems are being included in a single bCLEARer pipeline. In the initial 'in system' processing, the pipelines for the system's data should be kept separate - that is loosely coupled. When the data from the systems are being integrated, then this should be in a single cohesive unit. Pipeline gating helps with maintaining the balance between cohesion and coupling.

(see page 54)

1.6.7 Appendix - Pipeline Architecture Iconography

This appendix presents the iconography used to visualise the pipeline architecture. Visualisation is a vital tool for communicating and reviewing the architecture.

This appendix has two sections:

- Basic Pipeline Iconography (see page 58)
- bCLEARer Specific Pipeline Iconography (see page 67)

The first section presents the iconography common to pipeline architectures; and the second extends this with iconography specific to bCLEARer pipeline architectures.

1.6.7.1 Basic Pipeline Iconography

This section presents the iconography used to visualise the basic pipeline architecture.

This appendix has five sections:

- Filter-related icons (see page 59)
- Pipe-related icons (see page 60)
- Pipeline icons (see page 62)
- Nesting icons (see page 63)
- Pipeline architecture examples (see page 64)

The sections build the pipeline architecture iconography up from the atomic components (filter and pipe related) through the containers and finish with examples of uses of this iconography.

Each section is further broken down into sub-sections. And within these sub-sections there are tables of icons in alphabetical order.

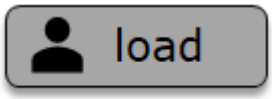
Where appropriate, there are links to examples of diagrams that illustrate the in-context use of the icons.

1.6.7.1.1 Filter-related icons

1.6.7.1.1.1 Components

iconography object names	iconography objects	notes
filter (data-type perspective)		This icon represents a filter. An example of its use is data type perspective (see page 64)
filter (process-type perspective)		This icon represents a filter. An example of its use is process type perspective (see page 64)
manual		This icon is used to adorn a filter, when manual work is involved.

1.6.7.1.1.2 Composites

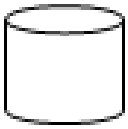
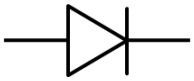
iconography object names	iconography objects	notes
filter (process-type perspective) & manual		This composite icon represents the “LOAD” bCLEARer stage which typically involves manual work. This illustrates the use of the manual icon as an adornment of a filter.

1.6.7.1.2 Pipe-related icons

1.6.7.1.2.1 Components

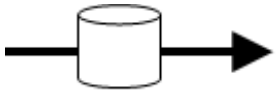
The following section presents the standard icon of a pipe that carries a dataset and its variant representations.
Icons in this list can be used on their own (see example given in the data type perspective (see page 64) glossary entry) or in conjunction with a pipe arrow icon (see example given in the process type perspective (see page 64) glossary entry).

They can optionally be named - see example given for the dataset (pipe) icon below.

iconography object names	iconography objects	notes
dataset (pipe)		This icon represents a pipe.
named dataset (pipe)		The dataset (pipe) icon (and its variants) can optionally be named.
dataset collection (pipe)		This icon represents a pipe that carries a collection of datasets. It can be used on its own (see example given in the data type perspective (see page 64) glossary entry) or in conjunction with a pipe arrow icon (see example given in the process type perspective (see page 64) glossary entry).
named dataset collection (pipe)		The dataset collection (pipe) icon (and its variants) can optionally be named.
gate		This icon represents a gate.

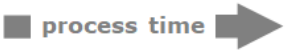
iconography object names	iconography objects	notes
pipe arrow		This icon represents a pipe as an arrow.

1.6.7.1.2.2 Composites

iconography object names	iconography objects	notes
dataset (pipe) & variants + pipe arrow		This composite icon represents a pipe that carries a dataset. The variant icons of the *dataset (pipe)* icon can similarly be combined with the pipe arrow.
dataset collection (pipe) + gate		This composite icon represents a gated dataset collection.
dataset collection (pipe) + pipe arrow		This composite icon represents a pipe that carries a dataset collection.

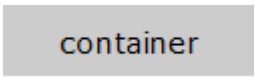
1.6.7.1.3 Pipeline icons

1.6.7.1.3.1 Components

iconography object names	iconography objects	notes
process time		This icon represents that during the pipeline process the data flows (in time) from left to right through the pipeline. It flows through components on the left before it flows on components to the right. See Process time (see page 7) for an example of use.

1.6.7.1.4 Nesting icons

1.6.7.1.4.1 Components

iconography object names	iconography objects	notes
container		See Nesting (see page 13) for examples.
encapsulation		See Nesting (see page 13) for examples.
sub-pipeline		See Nesting (see page 13) for examples.

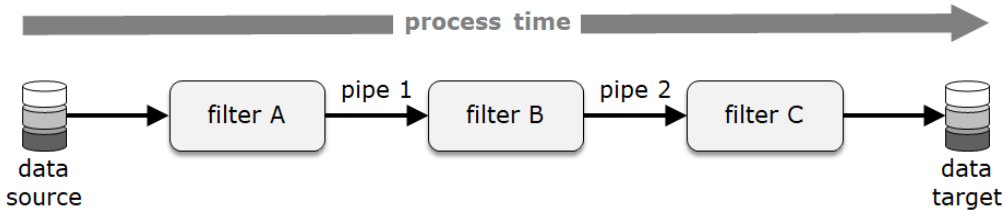
1.6.7.1.5 Pipeline architecture examples

1.6.7.1.5.1 Perspectives

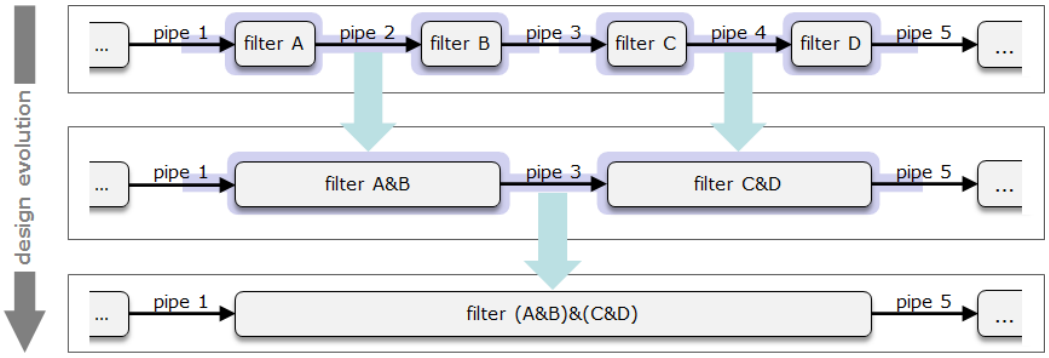
example names	examples
process type perspective	In a process type perspective, pipes are represented with the pipe icon (adorned or unadorned with a data icon) and filters are represented with the filter (process-type perspective) icon.  process type perspective example 1  process type perspective example 2
data type perspective	In a dataset type perspective, pipes are represented with the data icons (dataset collection icon or a dataset icon). Filters are represented with the filter (data-type perspective) icon.  data type perspective example

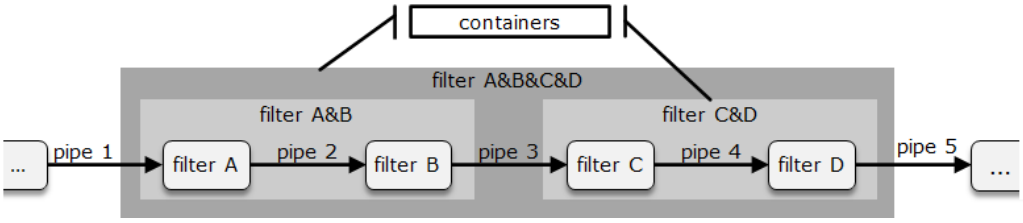
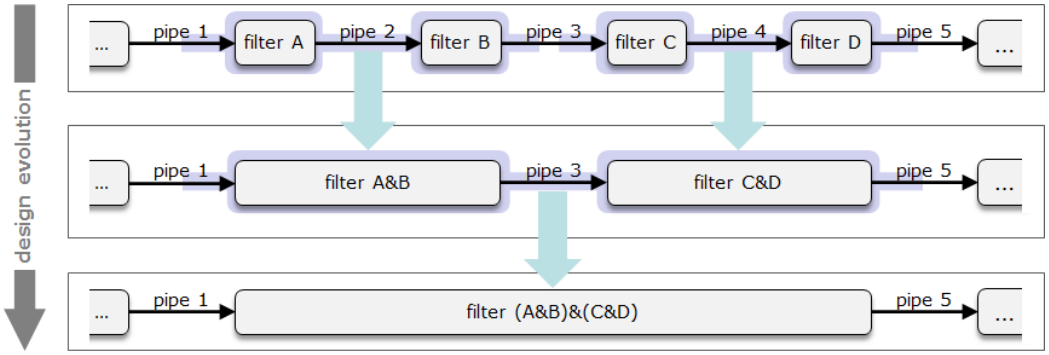
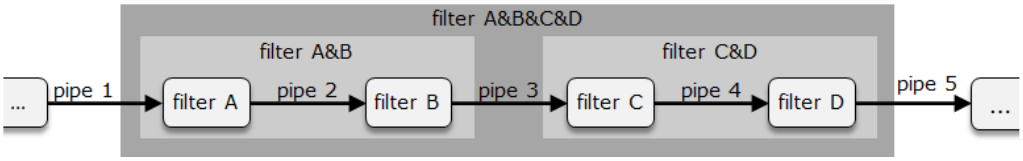
1.6.7.1.5.2 Simple pipeline

A simple pipeline diagram is obtained by combining Pipe-related icons (see page 60) and Filter-related icons (see page 59), according to one of the above-listed perspectives.
It may include a process time (see page 62) icon.

example names	examples
simple pipeline diagram (process type perspective)	
simple pipeline diagram (data type perspective)	

1.6.7.1.5.3 Nesting

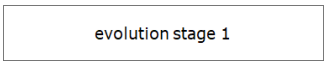
example names	examples
nested pipeline diagram	A nested pipeline diagram shows the evolution of a sub-pipeline into a nested pipeline; it represents the "external layer" of the pipeline at each stage. The pipeline chunk that encapsulates filter and pipe components at a stage in the evolution aligns with the encapsulated components from the previous stage. The diagram typically involves the evolution, sub-pipeline and encapsulation icons.  example

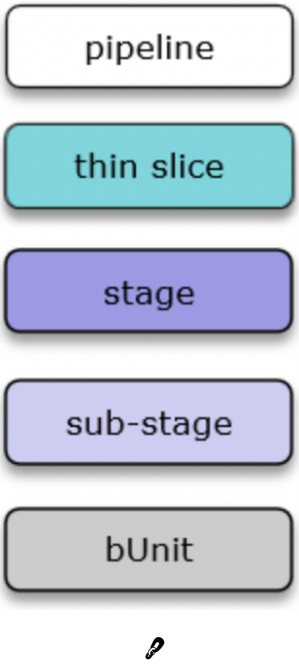
nesting diagram	A nesting diagram is a "see-through" representation of the different levels of nesting within a nested pipeline The diagram includes container icons.  example
corresponding nested pipeline and nesting diagrams	The nested pipeline diagrm and the nesting diagram provide complementary views on nesting. The nested pipeline diagram represents the "external layer" of the pipeline at each stage of the nesting evolution, while the nesting diagram represents the post-nesting stage only, with a view on the external layer and internal layer(s) of the pipeline. Below is an example of corresponding nested pipeline and nesting diagrams:  nested pipeline diagram example  nesting diagram example

1.6.7.2 bCLEARer Specific Pipeline Iconography

This section presents the iconography used to visualise the bCLEARer specific pipeline elements.

1.6.7.2.1 Components

iconography object names	iconography objects	notes
evolution stage		This icon is used to present a stage in the evolution of the pipeline flow. Typically, evolution stage icons are arranged in a vertical series to represent snapshots of the pipeline and used conjointly with the *evolutionary time* icon (see next entry). See Nesting (see page 13) for examples.
evolutionary time		This icon shows the direction of evolution of a pipeline (as pipes and filters may be added, removed - or changed, for instance). It is typically used conjointly with a series of *evolution stage* icon (see previous entry). The text may have a qualifying prefix, where a specific kind of evolution is being represented, for example, “design evolution”.

iconography object names	iconography objects	notes
nesting colouring		This palette is applied to the *filter (process-type perspective)* and the *container* icons to signify which of the five nesting levels the respective filter / container belongs to. See Architectural nesting breakdown (see page 21) for examples.

1.6.8 Appendix - Resilient Governance - The Right Balance of Principles and Rules

1.6.8.1 Overview

Any resilient system must strike a good balance between stability and flexibility. It must have stable elements that persist through change and flexible elements that can adapt to changes. From an ecosystem perspective, the stable infrastructure can be regarded as the environment in which the transformations unfold and evolve.

This good balance of stability and flexibility needs to be reflected in framing the governance within the right balance of principles and rules - particularly algorithmic rules. Where principles describe the broad aims and trust in the commitment of those executing them, whereas rules specify narrow, potentially rigid, constraints upon what can and cannot be done.

1.6.8.1.1 Rules

Rules aim to provide a structure that is, in theory, sufficient for resolving a particular type of case. They might seem to provide certainty and a clear standard of behaviour. They might seem to be easier to apply consistently. But it is difficult for rules to be congruent with their purpose, to achieve what it is they are intended to achieve. Rules are based upon a “best guess” as to the future. The rulemaker has to anticipate how the rule will be applied in the future. But new situations may arise that were not expected or known about when the rule was written. The rule may be interpreted and applied in ways that were not intended or anticipated by the writer. In practice, this not only leads to gaps, inconsistencies, rigidity, but is prone to degeneration into “creative compliance” (where the letter is followed but not the spirit). This creates a need for constant adjustment to new situations. This can then lead to a 'ratchet syndrome’, where more rules are created to address new problems or close new gaps, creating more gaps and so on.

1.6.8.1.2 Principles

Principles are typically more general and focus on purpose rather than process. This makes them more flexible without sacrificing congruence with their purpose. The execution of both rules and principles rely upon being understood, a shared understanding in the case of teams. However, in the case of principles there is typically a need for a wider and deeper understanding to interpret the principles in the way intended, or even suitably extend the interpretation to novel situations.

1.6.8.1.3 Principles and rules

Given the variety of situations in which bCLEARer is deployed and the requirement for rapid evolution and the associated radical change, rules by themselves are not the right tool for the job. Principles have a better initial fit for the bCLEARer situation. So there is no choice but to start with a principle-based approach. However, the requirement for a repeatable, scalable process means that in the end, the bCLEARer process needs to be developed as largely (computer-)executable rules.

The aim of this appendix is to describe the approach that underlies the bCLEARer approach to resilience.

1.6.8.2 Balancing Principles and Rules - A Literature View

Mature investigations into balancing of principles and rules appear in the literature for a number of domains, including the legal and regulatory domains. Among the aspects covered, there is one of interest to us - the simplicity or complexity of the topic to be covered. Unfortunately, there is not much investigation into the balancing of principles and rules in the IS domain, which we look at in the next section.

1.6.8.2.1 The need for a balance

Very generally, the difference between rules and principles is how tightly they constrain decision making - where rules impose stricter constraints than principles. It is accepted that both principles and rules are needed in most decision making. So the general concern is how to strike a balance between them. Whether or when the approach should be principle heavy and when rule heavy.

1.6.8.2.1.1 Simple or complex phenomena

(Braithwaite 2002 Rules and Principles) (see page 127) suggests that the simplicity or complexity of the phenomena makes a difference. Simple phenomena are more consistently regulated by precise rules than principles. However, as the regulated phenomena become more complex, principles deliver more consistency than rules. A central reason is that the iterative pursuit of precision in single rules increases the imprecision of a complex system of rules. By increasing the reliance we can place on a part of the law we reduce the reliability of the law as a whole.

As well as concerns about predictability (and so consistency) there are concerns about fairness. One such concern is how to define in advance all of the circumstances which should count as exceptions to the rule, particularly in complex situations. In the case of principles this kind of tight definition is not required.

Braithwaite suggests that consistency in complex domains can be even better realised by an appropriate mix of rules and principles than by principles alone. He further suggests that a key choice is between binding rules interpreted by non-binding principles and non-binding rules backed by binding principles. binding principles backing non-binding rules tend to regulate with greater certainty than principles alone and binding principles backing non-binding rules are more certain still if they are embedded in institutions of regulatory conversation that foster shared sensibilities.

(Schlag 1985 Rules and Standards) (see page 151) provides an example case which illustrates this issue:

> *In one torts casebook, for instance, Oliver Wendell Holmes and Benjamin Cardozo find themselves on opposite sides of a railroad crossing dispute. They disagree about what standard of conduct should define the obligations of a driver who comes to an unguarded railroad crossing. Holmes offers a rule: The driver must stop and look. Cardozo rejects the rule and instead offers a standard: The driver must act with reasonable caution. Which is the preferable approach? Holmes suggests that the requirements of due care at railroad crossings are clear and, therefore, it is appropriate to crystallize these obligations into a simple rule of law. Cardozo counters with scenarios in which it would be neither wise nor prudent for a driver to stop and look. Holmes might well have answered that Cardozo's scenarios are exceptions and that exceptions prove the rule. Indeed, Holmes might have parried by suggesting that the definition of a standard of conduct by means of a legal rule is predictable and certain, whereas standards and juries are not. This dispute could go on for quite some time.*

1.6.8.2.1.2 Other literature

These kinds of investigation into balancing are found in a number of others domains. For example, the accounting standards domain, (Bennett 2006 Rules, Principles and Judgments in Accounting Standards) (see page 125), (Benston 2006 Principles versus Rules-based Accounting Standards) (see page 126) and (Nelson 2003 Rules, Behavioral Evidence on the Effects of Principles- and Rules-Based Standards) (see page 137). This is an indicator of the generality of the concern.

(see page 70)

1.6.8.3 Balancing Principles and Rules in IS

Both principles and rules are deployed in the management of information systems (IS). However, there is little mature investigation on how to strike a balance between these. In this section, we give a flavour of the balance by looking at the principles and rules deployed in IT project management.

1.6.8.3.1 Rules

A traditional approach to IT project management is “regulation by rule”. The kinds of rules deployed are:

- detailed task orders
- deterministic project plans
- explicit role definitions
- timesheet-based project tracking

The assumption is that consistency is achieved through meticulous control and that any deviation from the rule (plan) is visible and immediately correctable.

pros	cons
lowering the cognitive burden	tighter constraints could make it so values is missed
stricter constraints mean less room for interpretation	make problem solving more difficult
consistency is easier to achieve	difficulty responding to change
rules allow for algorithmic decision making	lower trust
	less agile

1.6.8.3.2 Principles

The kind of principles used in IT project management are:

- continuous integration
- cross-functional teams
- short development iterations
- frequent releases of software

The assumption is that relying more on principles promotes agility and results in greater organizational success.

pros	cons
can benefit design by providing flexibility	requires judgment on how to apply a principle to a given context
greater range to navigate constraints	looser constraints mean more interpretive burden
promotes problem solving and innovation	
higher trust	
more agile	

1.6.8.3.3 Rules and Principles in IS

These examples of rules and principles in IS should give some idea of their ubiquity.

(see page 72)

1.6.8.4 Balancing Principles and Rules in bCLEARer

The nature of bCLEARer (and BORO) work requires the flexibility, innovation and agility that principles promote, hence a principle-heavy balance is adopted. Examples of the kind of principles in bCLEARer are:

- Aim for a manageable-sized set of rules and constraints
- Ensure full trackability and traceability
- Take small steps in the process when evolving data
- Iterate the bCLEARer stages

(see page 74)

1.6.9 Appendix - Separation of Concerns Principle

Separation of Concerns is one of the guiding principles of bCLEARer (see Appendix - Resilient Governance - The Right Balance of Principles and Rules (see page 69)).

1.6.9.1 History

The phrase 'separation of concerns' was coined within computing by Edsger W. Dijkstra in (Dijkstra 1974 On the role of scientific thought) (see page 130). He suggested that adopting this principle would simplify the development and maintenance of scientific classification, as it treats independent aspects independently and so avoids creating unnecessary dependencies. More specifically, he thought when concerns are well-separated, individual sections can be reused, developed and updated independently.

The idea was well received. For example, a decade and a half later, in 1989, Chris Reade's description of the principle in (Reade 1989 Elements of Functional Programming) (see page 149) showed that it was an accepted idea. Reade describes separation of concerns saying:

> *The programmer is having to do several things at the same time, namely,*
>
> 1. describe what is to be computed;
> 2. organise the computation sequencing into small steps;
> 3. organise memory management during the computation.

he continues,

> *Ideally, the programmer should be able to concentrate on the first of the three tasks (describing what is to be computed) without being distracted by the other two, more administrative, tasks. Clearly, administration is important, but by separating it from the main task we are likely to get more reliable results and we can ease the programming problem by automating much of the administration.*
>
> *The separation of concerns has other advantages as well. For example, program proving becomes much more feasible when details of sequencing and memory management are absent from the program. Furthermore, descriptions of what is to be computed should be free of such detailed step-by-step descriptions of how to do it, if they are to be evaluated with different machine architectures. Sequences of small changes to a data object held in a store may be an inappropriate description of how to compute something when a highly parallel machine is being used with thousands of processors distributed throughout the machine and local rather than global storage facilities.*
>
> *Automating the administrative aspects means that the language implementor has to deal with them, but he/she has far more opportunity to make use of very different computation mechanisms with different machine architectures.*

Thus he gives a clear motivation for separating administration concerns from concerns about 'what is to be computed'.

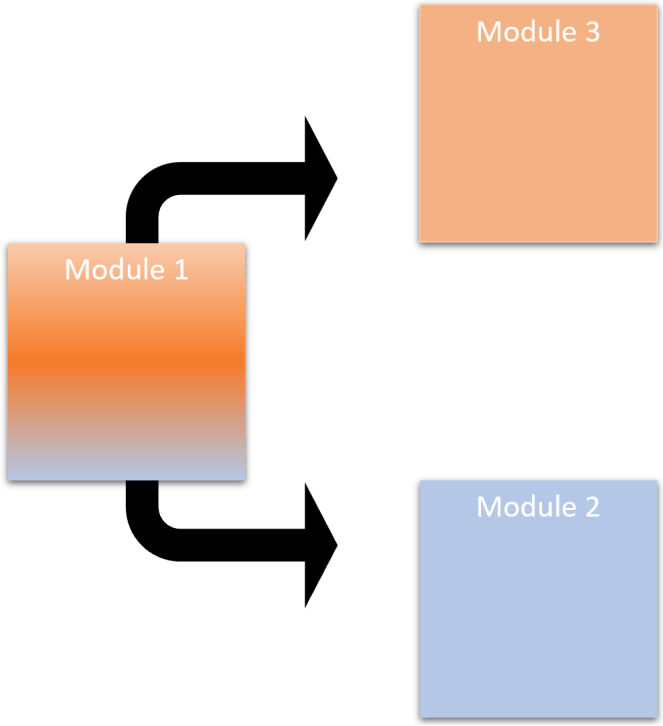
1.6.9.2 What is a concern?

In computer science, a concern is a particular set of information that has an effect on the code of a computer program. A concern can be as general as "the details of the hardware for an application", or as specific as

"the name of which class to instantiate". Generally, a concern is an aspect in designing a system that requires developer attention.

1.6.9.3 What is the principle?

The separation of concerns principle, generally, directs that independent aspects of a system should be treated independently by developers. For instance, if a system has two functions, and they operate on different things, then they should be in different modules. This can be seen as the simple pattern shown graphically below:



separating two concerns into two modules

It can also be accomplished by encapsulation (aka information hiding), layered (nested) designs - for example, presentation layer, business logic layer, data access layer, persistence layer. This notion extends to areas outside of IT, including urban planning, architecture and information design.

1.6.9.3.1 An Example

> Separation of concerns is crucial to the design of the Internet. In the Internet Protocol Suite[1], great efforts have been made to separate concerns into well-defined layers[2]. This allows protocol designers to focus on the concerns in one layer, and ignore the other layers. The Application Layer protocol SMTP, for example, is concerned about all the details of conducting an email session over a reliable transport service (usually TCP[3]), but not in the least concerned about how the transport service makes that service reliable. Similarly, TCP is not concerned about the routing of data packets, which is handled at the Internet Layer[4].

1.6.9.3.2 The value of separation of concerns

The principle has a range of values, including:

- avoids the creation of unnecessary dependencies, thus promoting the modularity of a system
- promotes encapsulation, which is a means of information hiding[5]

For computer programs it:

- increases modularity
- increases the reusability of modules
- simplifies development and maintenance
    - make modules easier to understand by coders
    - makes them easier to test

It also supports other related principles, including:

- treating independent aspects independently leads to looser coupling between modules (see Appendix - Loose Coupling and Tight Cohesion Principle (see page 54))
- promotes development of SSOT Appendix - Single-Transformation (Responsibility) Principle (STP) (see page 78)

1.6.9.3.3 In BORO and bCLEARer

The separation of concerns is a central principle in both BORO and bCLEARer. It is a key principle of BORO Clean Coding Principles (see page 104) as well as the pipeline architecture framework.

(see page 75)

1 https://en.wikipedia.org/wiki/Internet_Protocol_Suite
2 https://en.wikipedia.org/wiki/OSI_Model
3 https://en.wikipedia.org/wiki/Transmission_Control_Protocol
4 https://en.wikipedia.org/wiki/Internet_Layer
5 https://en.wikipedia.org/wiki/Information_hiding

1.6.10 Appendix - Single-Transformation (Responsibility) Principle (STP)

1.6.10.1 History

The related term 'Single Responsibility Principle' (SRP) was introduced by Robert C. Martin in his article (Martin 2005 The Principles of OOD) (see page 133) as part of his *Principles of Object Oriented Design*,[5][6] where it was stated as: “A class should have one, and only one, reason to change.” It was made popular in (Martin 2003 Agile Software Development) (see page 132). Martin described it as being based on the principle of cohesion, as described in (DeMarco 1979 Structured Analysis and System Specification) (see page 129) and (Page-Jones 1988 The Practical Guide to Structured Systems Design) (see page 138).

People were querying what the phrase "reason for change" meant. Martin published a blog post (Martin 2014 The Single Responsibility Principle) (see page 135) to clarify this. He said:

“Another wording for the Single Responsibility Principle is: Gather together the things that change for the same reasons. Separate those things that change for different reasons.”

and

“However, as you think about this principle, remember that the reasons for change are *people*. It is *people* who request changes. And you don’t want to confuse those people, or yourself, by mixing together the code that many different people care about for different reasons.”

More recently, (Martin 2018 Clean Architecture) (see page 136) stated that: A module should be responsible to one, and only one, actor. Where the term actor refers to a group (consisting of one or more stakeholders or users) that requires a change in the module.

The current view is that SRP aims to add some specific granularity to the Appendix - Separation of Concerns Principle (see page 75) by saying how the class units of OO code should be separated.

1.6.10.2 Transformation

In bCLEARer, the equivalent of a class is a transformation. So the the Single-Transformation (Responsibility) Principle (STP) is the analogue of SRP principle for bCLEARer transformations. The principle suggests that the transformations should be architected so that they deal with one concern for one actor.

(see page 78)

6 https://en.wikipedia.org/wiki/Single_responsibility_principle#cite_note-5

1.6.11 Appendix - Historic Pipeline Examples

This appendix describes historic pipeline examples in the following two sections:

Code Examples
A sample of historic bCLEARer project code examples that use the Library Examples (see page 87).

Library Examples
A sample of libraries developed by BORO that have been used by historic bCLEARer project Code Examples (see page 80).

1.6.11.1 Code Examples

A sample of historic bCLEARer project code examples that use the Library Examples (see page 87).

- UNICLASS bCLEARer (see page 81)
- bOSON 1.1 - Surface Onomatology (see page 83)
- bOSON 1.2 - Deep Onomatology (Coordinates) (see page 85)

The descriptions of these examples took into account the characteristics noted in bCLEARer Profiling Characteristics Lite (see page 202).

These projects are currently closed.

1.6.11.1.1 UNICLASS bCLEARer

Example - bCLEARer Process - UNICLASS bCLEARer

The UNICLASS bCLEARer Process[7] GitHub repository contains an example of the bCLEARer process (see bCLEARer - an introduction (see page 2)) applied implemented in Python that has been applied to the UNICLASS classification dataset[8].

The Python code transparently documents the transformations that arise from the data cleaning and ontological analysis that takes place in the different stages of the process, and it outputs the result in a self-contained folder system divided into sub-folders corresponding to each of the implemented bCLEARer stages and sub-stages.

As a standard bCLEARer inspection practice, output is provided for each of the sub-stages. This enables the user to check the progress of the process.

For each of the sub-stages, the user can check the progress of the analysis in three kinds of output files:

- domain tables: collection of data tables showing a standard human-readable structure (as CSV and Access databases)
- nf ea com tables: collection of data tables which are the result of processing the domain tables to give them a more UML-friendly structure (influenced by the Enterprise Architect Data Model) - (as CSV and Access databases)
- Enterprise Architect[9] model files - useful for visualizing

This project applied the Architectural nesting breakdown (see page 21). This nesting architecture was developed using standard code patterns to facilitate rapid evolutionary development.

The pipeline experienced rapid evolution. However, only the latest snapshot was copied to the Open Source repo, so the evolution doesn't show in the repo's history.

The latest snapshot has a single bCLEARer sequence implementing the Stages level (see page 26) level pattern for Collect, Load and Evolve.

The Collect stage data (the UNICLASS classification dataset) can be downloaded online here[10]. The results of early Load are stored in the repository's Load resources folder[11].

The repo contains code for the Load (bCLEARer Load orchestrator[12]) and Evolve (bCLEARer Evolve orchestrator[13]) stages.

7 https://github.com/boro-alpha/uniclass_to_nf_ea_com
8 https://www.thenbs.com/our-tools/uniclass-2015
9 https://sparxsystems.com/products/ea/
10 https://www.thenbs.com/our-tools/uniclass-2015
11 https://github.com/boro-alpha/uniclass_to_nf_ea_com/tree/master/uniclass_to_nf_ea_com_source/c_resources/1_load_inputs
12 https://github.com/boro-alpha/uniclass_to_nf_ea_com/blob/master/uniclass_to_nf_ea_com_source/b_code/orchestrators/stages/load/load_stage_4_orchestrator.py
13 https://github.com/boro-alpha/uniclass_to_nf_ea_com/blob/master/uniclass_to_nf_ea_com_source/b_code/orchestrators/uniclass_bclearer_orchestrator.py

The bCLEARer sub-stages orchestrate a series of atomic single-purposed bUnits level (see page 27) (this Evolve substage code[14] shows a sequence of bUnit operations).

Object identity is maintained using UUIDS complying with the RFC 4122[15] standard (we can see the creation of a UUID for a new object in this Evolve substage code example[16]) which allows the tracking of objects throughout the process.

Object identity based bCLEARer Universes (see page 103) (NfEaComUniverses[17] class) are used to store and transport data. Snapshots of the project's universe are taken in specific points to maximise inspectability (example here[18]).

There are object identity universe-based **gates** at the end of bCLEARer stages and sub-stages (as we can see in this EVOLVE substage code example[19]). This enabled common code to be used to visualise the gate snapshots of the universes and so inspect the process at each stage (common bCLEARer substage visualizer[20]).

This project did not require bCLEARer Query Architecture (see page 100) or reporting.

A significant proportion of the code is reused from these **BORO common libraries**: NF Common (see page 93), NF EA Common Tools (see page 89), EA Interop Service (see page 95).

As is standard BORO practice, this project applied BORO Clean Coding Principles (see page 104).

This project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

(see page 208)

14 https://github.com/boro-alpha/uniclass_to_nf_ea_com/blob/master/uniclass_to_nf_ea_com_source/b_code/migrators/uniclass_raw_to_domain/evolve/evolve_stage_2/domain_tables_data_processor/evolve_stage_2_domain_tables_getter.py

15 https://datatracker.ietf.org/doc/html/rfc4122

16 https://github.com/boro-alpha/uniclass_to_nf_ea_com/blob/753e97467ce53c25bc86341b915489c2eeeb3f49/uniclass_to_nf_ea_com_source/b_code/migrators/uniclass_raw_to_domain/evolve/evolve_stage_5/domain_tables_data_processor/evolve_stage_5_domain_tables_getter.py

17 https://github.com/boro-alpha/nf_ea_common_tools/blob/master/nf_ea_common_tools_source/b_code/services/general/nf_ea/com/nf_ea_com_universes.py

18 https://github.com/boro-alpha/uniclass_to_nf_ea_com/blob/master/uniclass_to_nf_ea_com_source/b_code/migrators/uniclass_nf_ea_com_exporters/export_nf_ea_com_orchestrator.py

19 https://github.com/boro-alpha/uniclass_to_nf_ea_com/blob/master/uniclass_to_nf_ea_com_source/b_code/orchestrators/stages/evolve/evolve_stage_4_orchestrator.py

20 https://github.com/boro-alpha/bclearer/blob/master/bclearer_source/b_code/substages/visualizations/instrumentation_and_visualization_runner.py

1.6.11.1.2 bOSON 1.1 - Surface Onomatology

Example - bCLEARer Process - bOSON 1.1 (bCLEARer Ordnance Survey Open Names) Surface Onomatology

The bOSON 1.1 (bCLEARer Ordnance Survey Open Names) Surface Onomatology[21] GitHub repository contains an example of the bCLEARer process implemented in Python that has been applied to the Ordnance Survey Open Names[22] dataset. At a more general level, this code provides an example of how to automatically implement (at scale) a surface onomatology pattern over structured data.

The Python code transparently documents the transformations that arise from the data cleaning and ontological analysis that takes place in the different stages of the bCLEARer process, and it outputs the result in a self-contained folder system divided into sub-folders for each of the bCLEARer stages and sub-stages.

As a standard bCLEARer inspection practice, output was provided for each of the sub-stages. This enabled the user to check the progress of the process.

This project applied the Architectural nesting breakdown (see page 21). This nesting architecture was developed using standard code patterns to facilitate rapid evolutionary development.

The pipeline experienced rapid evolution. However, only the latest snapshot was copied to the Open Source repo, so the evolution doesn't show in the repo's history.

The latest snapshot has a single bCLEARer sequence implementing the Stages level (see page 26) level pattern for Collect, Load and Evolve.

The Collect stage data (OS Open Names GML source data) can be downloaded online here[23].

The repo contains code for the Load (bCLEARer Load orchestrator[24]) and Evolve (bCLEARer Evolve orchestrator[25]) stages.

The bCLEARer sub-stages orchestrate a series of atomic single-purposed bUnits level (see page 27) (this Load substage code[26] shows a sequence of bUnit operations).

Object identity is maintained using UUIDS complying with the RFC 4122[27] standard (in this Load bUnit code example[28] we can see how a new object is given a UUID) which allows the tracking of objects throughout the process.

21 https://github.com/boro-alpha/bclearer_boson_1_1
22 https://www.ordnancesurvey.co.uk/products/os-open-map-local
23 https://osdatahub.os.uk/downloads/open/OpenNames
24 https://github.com/boro-alpha/bclearer_boson_1_1/blob/master/bclearer_boson_1_1_source/b_code/orchestrators/boson_1_load_stage_orchestrator.py
25 https://github.com/boro-alpha/bclearer_boson_1_1/blob/master/bclearer_boson_1_1_source/b_code/orchestrators/boson_1_evolve_stage_orchestrator.py
26 https://github.com/boro-alpha/bclearer_boson_1_1/blob/master/bclearer_boson_1_1_source/b_code/substages/visualization_substages/boson_1_visualization_substage_load_gml_data_runner.py
27 https://datatracker.ietf.org/doc/html/rfc4122
28 https://github.com/boro-alpha/bclearer_boson_1_1/blob/master/bclearer_boson_1_1_source/b_code/substages/operations/a_load/boson_1_gml_data_loader/os_open_names_domain/os_open_names_appenders/b1_os_open_names_dictionary_appender.py

Object identity based bCLEARer Universes (see page 103) (NfEaComUniverses[29] class) are used to store and transport data. Snapshots of the project's universe are taken after each bUnit to maximise inspectability (as we can see in this Evolve substage code example[30]).

There are object identity universe-based **gates** at the end of bCLEARer stages and sub-stages (as we can see in this Evolve substage code example[31]). This enabled common code to be used to visualise the gate snapshots of the universes and so inspect the process at each stage (common bCLEARer substage visualizer[32]).

This project did not require bCLEARer Query Architecture (see page 100) or reporting.

A significant proportion of the code is reused from these **BORO common libraries**: NF Common (see page 93), NF EA Common Tools (see page 89), EA Interop Service (see page 95), bCLEARer (see page 88), BNOP (see page 97), NF EA Com BNOP (see page 91), BORO Common (see page 94).

As is standard BORO practice, this project applied BORO Clean Coding Principles (see page 104).

This project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

(see page 204)

29 https://github.com/boro-alpha/nf_ea_common_tools/blob/master/nf_ea_common_tools_source/b_code/services/general/nf_ea/com/nf_ea_com_universes.py
30 https://github.com/boro-alpha/bclearer_boson_1_1/blob/master/bclearer_boson_1_1_source/b_code/substages/visualization_substages/boson_1_visualization_substage_separate_names_and_instances_runner.py
31 https://github.com/boro-alpha/bclearer_boson_1_1/blob/master/bclearer_boson_1_1_source/b_code/substages/visualization_substages/boson_1_visualization_substage_separate_names_and_instances_runner.py
32 https://github.com/boro-alpha/bclearer/blob/master/bclearer_source/b_code/substages/visualizations/instrumentation_and_visualization_runner.py

1.6.11.1.3 bOSON 1.2 - Deep Onomatology (Coordinates)

Example - bCLEARer Process - bOSON 1.2 (bCLEARer Ordnance Survey Open Names) Deep Onomatology - Coordinates

The bOSON 1.2 (bCLEARer Ordnance Survey Open Names) Deep Onomatology[33]GitHub repository contains an example of the bCLEARer process implemented in Python that has been applied to the Ordnance Survey Open Names[34] dataset and the INSPIRE[35] metamodel. At a more general level, this code provides an example of how to automatically implement (at scale) a deep onomatology pattern over structured data.

The Python code transparently documents the transformations that arise from the data cleaning and ontological analysis that takes place in the different stages of the bCLEARer process, and it outputs the result in a self-contained folder system divided into sub-folders corresponding to each of the bCLEARer stages and sub-stages.

As a standard bCLEARer inspection practice, output was provided for each of the sub-stages. This enabled the user to check the progress of the process.

This project applied the Architectural nesting breakdown (see page 21). This nesting architecture was developed using standard code patterns to facilitate rapid evolutionary development.

The pipeline experienced rapid evolution. However, only the latest snapshot was copied to the Open Source repo, so the evolution doesn't show in the repo's history.

The latest snapshot has a single bCLEARer sequence implementing the Stages level (see page 26) level pattern for Collect, Load and Evolve.

The Collect stage data can be downloaded online; the OS Open Names GML from here[36], and the INSPIRE metamodel source data from here[37].

The repo contains code for the Load (bCLEARer Load orchestrator[38]) and Evolve (bCLEARer Evolve orchestrator[39]) stages.

The bCLEARer sub-stages orchestrate a series of atomic single-purposed bUnits level (see page 27) (this Evolve substage code[40] shows a sequence of bUnit operations).

Object identity is maintained using UUIDS complying with the RFC 4122[41] standard (example here[42]) which allows the tracking of objects throughout the process.

33 https://github.com/boro-alpha/bclearer_boson_1_2
34 https://www.ordnancesurvey.co.uk/products/os-open-map-local
35 https://inspire.ec.europa.eu/
36 https://osdatahub.os.uk/downloads/open/OpenNames
37 https://inspire.ec.europa.eu/portfolio/data-models
38 https://github.com/boro-alpha/bclearer_boson_1_2/blob/master/bclearer_boson_1_2_source/b_code/orchestrators/boson_1_2_load_stage_orchestrator.py
39 https://github.com/boro-alpha/bclearer_boson_1_2/blob/master/bclearer_boson_1_2_source/b_code/orchestrators/boson_1_2_evolve_stage_orchestrator.py
40 https://github.com/boro-alpha/bclearer_boson_1_2/blob/master/bclearer_boson_1_2_source/b_code/substages/visualization_substages/boson_1_2_visualization_substage_expose_implicit_point_structure_and_names_runner.py
41 https://datatracker.ietf.org/doc/html/rfc4122
42 https://github.com/boro-alpha/bclearer_boson_1_2/blob/master/bclearer_boson_1_2_source/b_code/substages/operations/b_evolve/coordinate_lines/common/nf_uuid_keyed_on_name_prefix_dictionary_getter.py

Object identity based bCLEARer Universes (see page 103) (NfEaComUniverses[43] class) are used to store and transport data. Snapshots of the project's universe are taken in specific points to maximise inspectability (as we can see in this EVOLVE substage code example[44]).

There are object identity universe-based **gates** at the end of bCLEARer stages and sub-stages (as we can see in this EVOLVE substage code example[45]). This enabled common code to be used to visualise the gate snapshots of the universes and so inspect the process at each stage (common bCLEARer substage visualizer[46]).

This project did not require bCLEARer Query Architecture (see page 100) or reporting.

A significant proportion of the code is reused from these **BORO common libraries**: NF Common (see page 93), NF EA Common Tools (see page 89), EA Interop Service (see page 95), bCLEARer (see page 88), BNOP (see page 97), NF EA Com BNOP (see page 91), BORO Common (see page 94).

As is standard BORO practice, this project applied BORO Clean Coding Principles (see page 104).

This project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

(see page 206)

43 https://github.com/boro-alpha/nf_ea_common_tools/blob/master/nf_ea_common_tools_source/b_code/services/general/nf_ea/com/nf_ea_com_universes.py
44 https://github.com/boro-alpha/bclearer_boson_1_2/blob/master/bclearer_boson_1_2_source/b_code/substages/visualization_substages/boson_1_2_visualization_substage_coordinate_lines_runner.py
45 https://github.com/boro-alpha/bclearer_boson_1_2/blob/master/bclearer_boson_1_2_source/b_code/substages/visualization_substages/boson_1_2_visualization_substage_generalise_runner.py
46 https://github.com/boro-alpha/bclearer/blob/master/bclearer_source/b_code/substages/visualizations/instrumentation_and_visualization_runner.py

1.6.11.2 Library Examples

A sample of libraries developed by BORO that have been used by historic bCLEARer project Code Examples (see page 80).

- bCLEARer (see page 88)
- NF EA Common Tools (see page 89)
- NF EA Com BNOP (see page 91)
- NF Common (see page 93)
- BORO Common (see page 94)
- EA Interop Service (see page 95)
- BNOP (see page 97)

All code here was developed using BORO Clean Coding Principles (see page 104).

These are historic snapshots of working libraries that are under continual development.

1.6.11.2.1 bCLEARer

Example - BORO Code Library - bCLEARer

This is a historic snapshot of a working library that is under continual development.

This bCLEARer[47] GitHub repository is a Python reference library containing utility functions that are being used by other GitHub projects within the bCLEARer process developed by BORO Solutions[48].

Examples of the common code are:

- operations[49] - used in the execution of bUnits level (see page 27)
- visualizations[50] - used for gate inspection.

The code was developed using BORO Clean Coding Principles (see page 104).

This GitHub project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

(see page 210)

47 https://github.com/boro-alpha/bclearer
48 https://borosolutions.net/
49 https://github.com/boro-alpha/bclearer/tree/master/bclearer_source/b_code/substages/operations
50 https://github.com/boro-alpha/bclearer/tree/master/bclearer_source/b_code/substages/visualizations

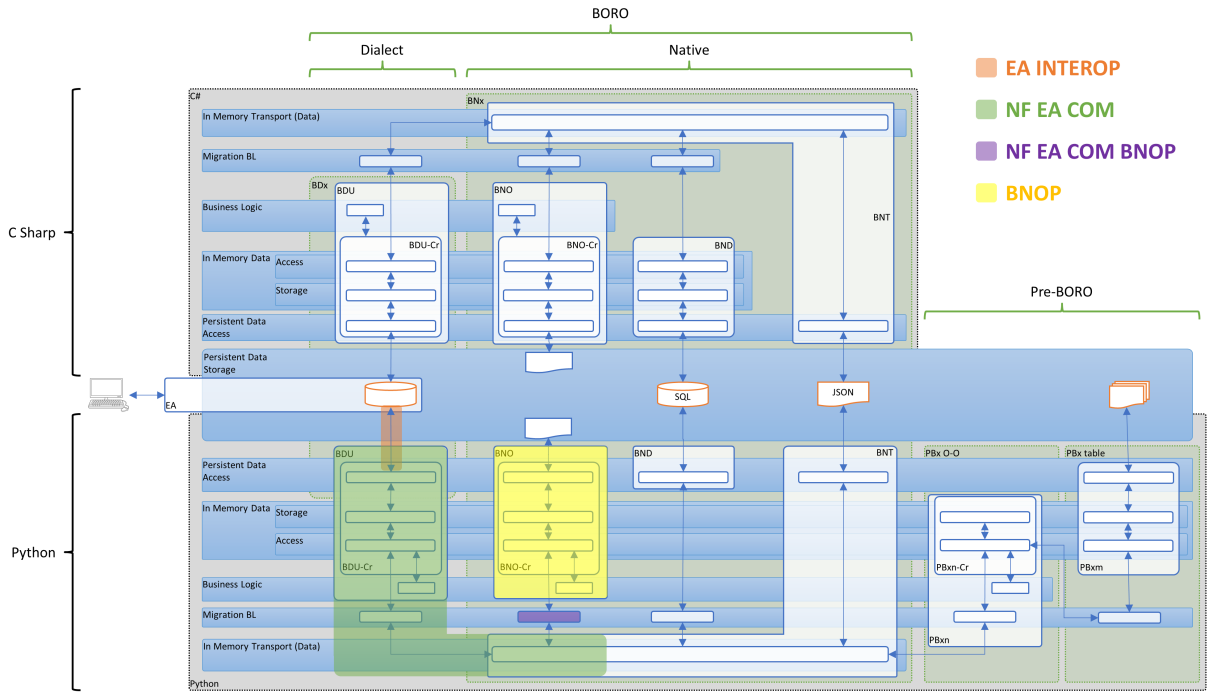
1.6.11.2.2 NF EA Common Tools

Example - BORO Code Library - NF EA Common Tools

This is a historic snapshot of a working library that is under continual development.

This NF EA Common Tools[51] GitHub repository is a Python reference library containing utility functions that are being used by other GitHub projects within the NF EA domain.

The historic NF architecture diagram below highlights the footprint of the NF EA COM (the main component of the NF EA Common Tools Library) along with the other libraries.



historic nf architecture - with library footprints

The code was developed using BORO Clean Coding Principles (see page 104).

This GitHub project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

51 https://github.com/boro-alpha/nf_ea_common_tools

(see page 218)

1.6.11.2.3 NF EA Com BNOP

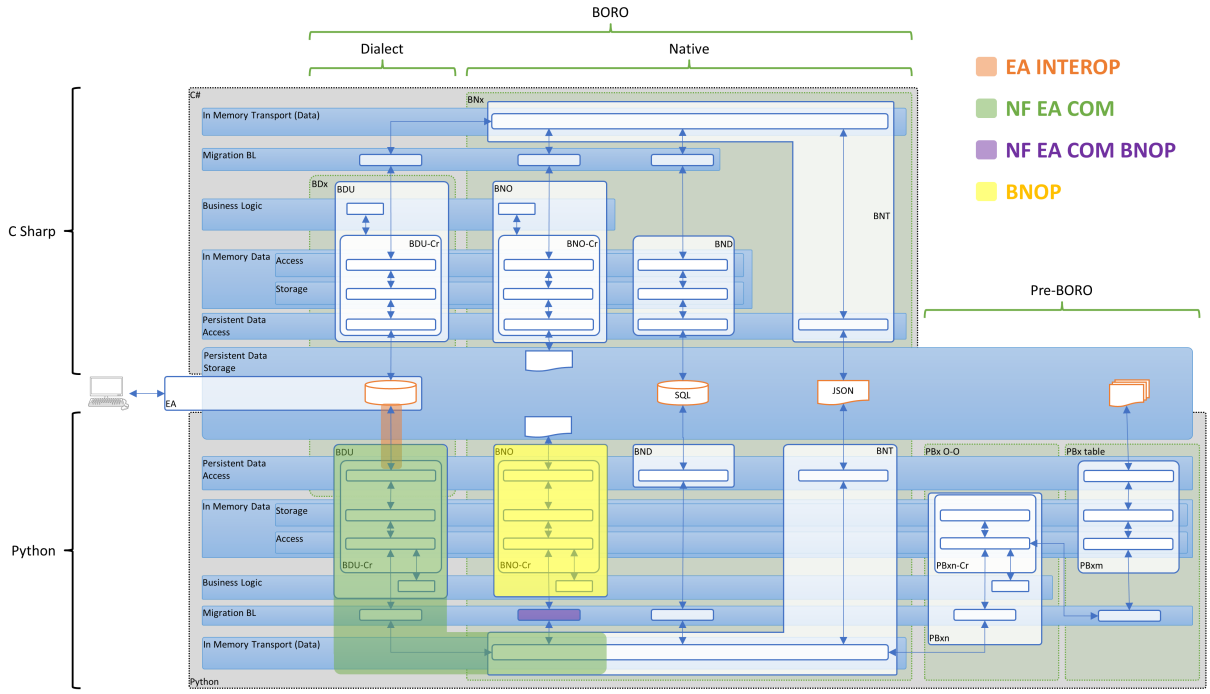
Example - BORO Code Library - NF EA COM BNOP

This is a historic snapshot of a working library that is under continual development.

This NF EA COM BNOP[52] GitHub repository is a Python reference library containing utility functions that migrate between the NF EA COM domain and the BNOP[53] domain.

- The NF EA COM domain is an inter-domain data layer providing objects and functions to store the data in a format that aligns closely with EA (Enterprise Architect[54]). It can then be easily imported into and visualised.
- BNOP is the boro native objects (in Python)

The historic NF architecture diagram below highlights the footprint of this NF EA COM BNOP library along with the other libraries.



historic nf architecture - with library footprints

The code in this library is being used by other GitHub projects within the BORO domain.

52 https://github.com/boro-alpha/nf_ea_com_bnop
53 https://github.com/boro-alpha/bnop
54 https://sparxsystems.com/products/ea/

The code was developed using BORO Clean Coding Principles (see page 104).

This GitHub project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

(see page 216)

1.6.11.2.4 NF Common

Example - BORO Code Library - NF Common

This is a historic snapshot of a working library that is under continual development.

This (New Foundations[55]) NF Common[56] GitHub repository is a Python reference library containing utility functions that are being used by other GitHub projects within the BORO domain.

The code was developed using BORO Clean Coding Principles (see page 104).

This GitHub project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

(see page 215)

55 https://en.wikipedia.org/wiki/New_Foundations
56 https://github.com/boro-alpha/nf_common

1.6.11.2.5 BORO Common

Example - BORO Code Library - BORO Common

This is a historic snapshot of a working library that is under continual development.

This BORO Common[57] GitHub repository is a Python reference library containing enumerations that are being used by other GitHub projects within the BORO domain.

The code was developed using BORO Clean Coding Principles (see page 104).

This GitHub project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

(see page 212)

57 https://github.com/boro-alpha/boro_common

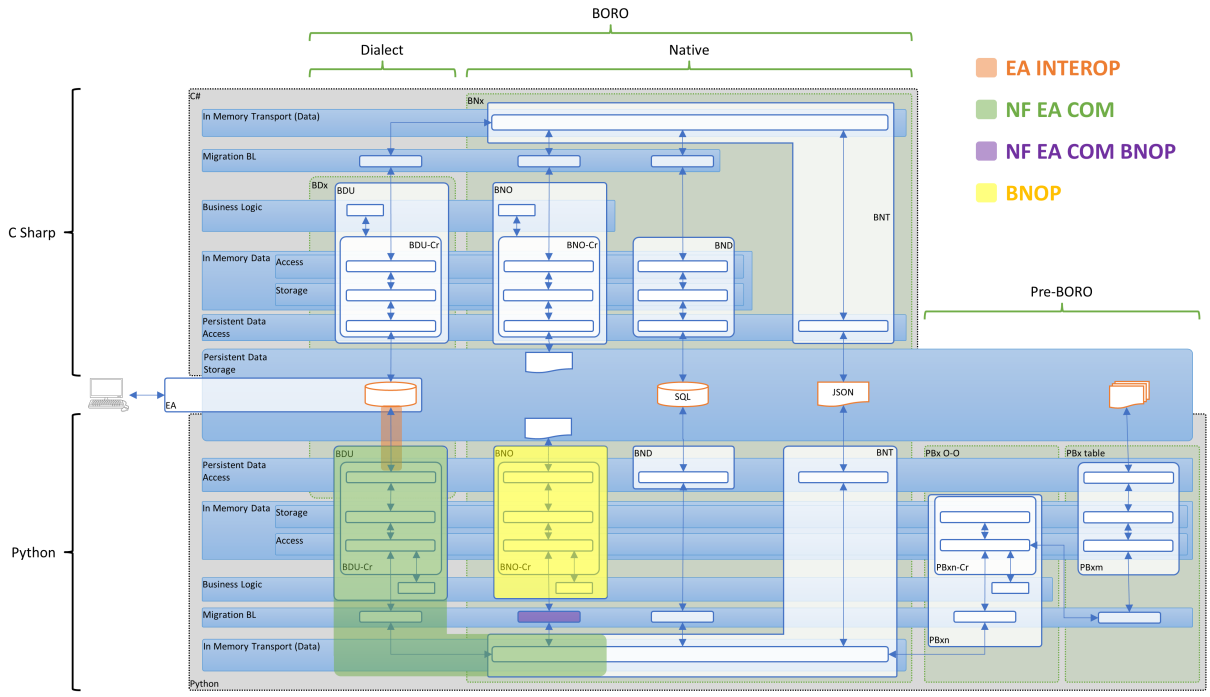
1.6.11.2.6 EA Interop Service

Example - BORO Code Library - EA Interop Service

This is a historic snapshot of a working library that is under continual development.

This EA Interop Service[58] GitHub repository is a Python reference library that exposes parts of the Sparx EA object model[59] interface developed by BORO Solutions[60].

The historic NF architecture diagram below highlights the footprint of this EA Interop Service library along with the other libraries.



historic nf architecture - with library footprints

The code in this library is being used by other GitHub projects within the BORO domain.

The code was developed using BORO Clean Coding Principles (see page 104).

58 https://github.com/boro-alpha/ea_interop_service
59 https://sparxsystems.com/enterprise_architect_user_guide/14.0/automation/theautomationinterface.html
60 https://borosolutions.net/

This GitHub project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

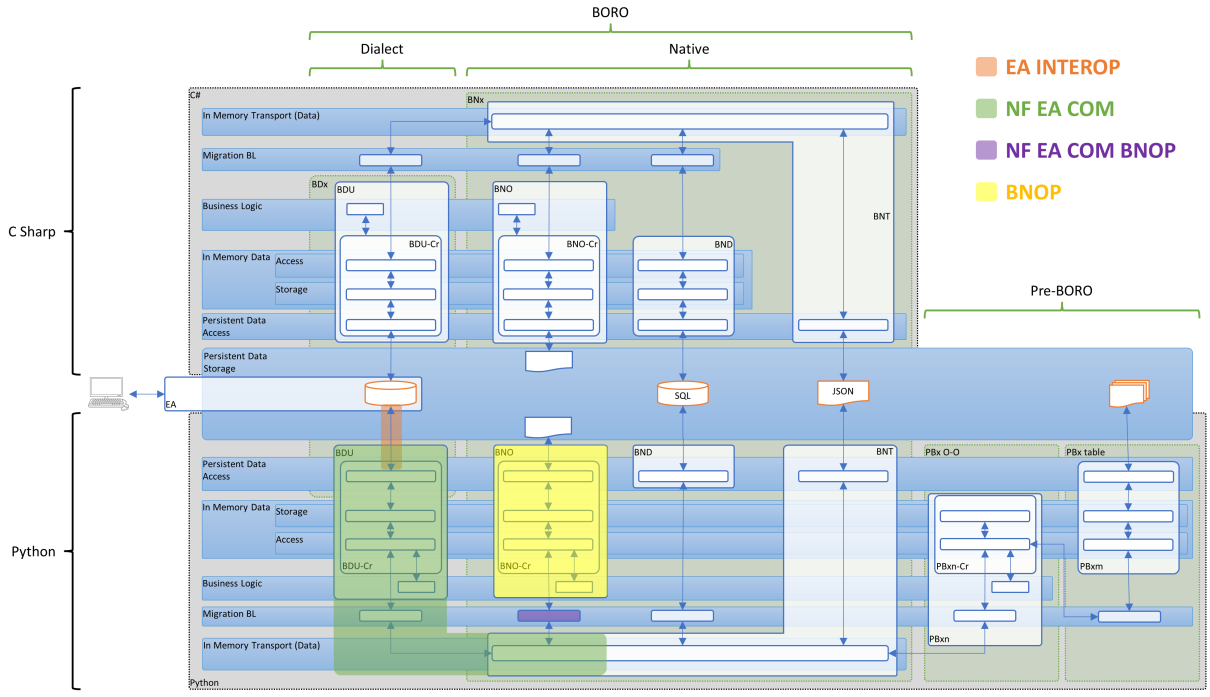
(see page 213)

1.6.11.2.7 BNOP

Example - BORO Code Library - BNOP

This is a historic snapshot of a working library that is under continual development.

This BNOP[61] GitHub repository is a Python reference library containing the code for the BNOP Domain (BORO Native Objects (Python)) - this is an implementation of the BORO Top Ontology in Python. This has been implemented in a range of languages - as an historic NF architecture diagram below shows (the footprint of this BNOP library is highlighted - as are the other libraries).



historic nf architecture - with library footprints

The code in this library is being used by other GitHub projects within the BORO domain.

The code was developed using BORO Clean Coding Principles (see page 104).

61 https://github.com/boro-alpha/bnop

This GitHub project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

(see page 211)

1.6.12 Appendix - Further Related Topics

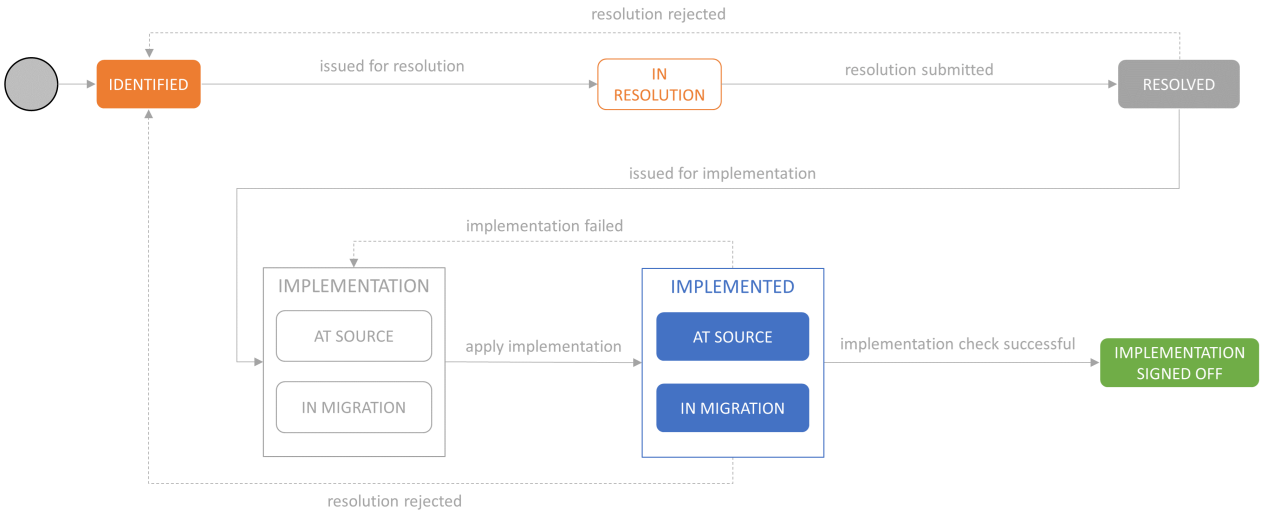
This appendix deals briefly with a range of further related topics, which are possible candidates for future eManuals. The topics are:

1.6.12.1 bCLEARer Query Architecture

At the start of a typical bCLEARer project we do not know much about how the data needs to be cleaned and transformed. One of the aims of the project is to make explicit concerns about this data - which can then be investigated and resolved. One of the ways the bCLEARer process makes concerns explicit is using *queries*. Hence the design of the architecture of these is an important aspect of the overall bCLEARer architecture.

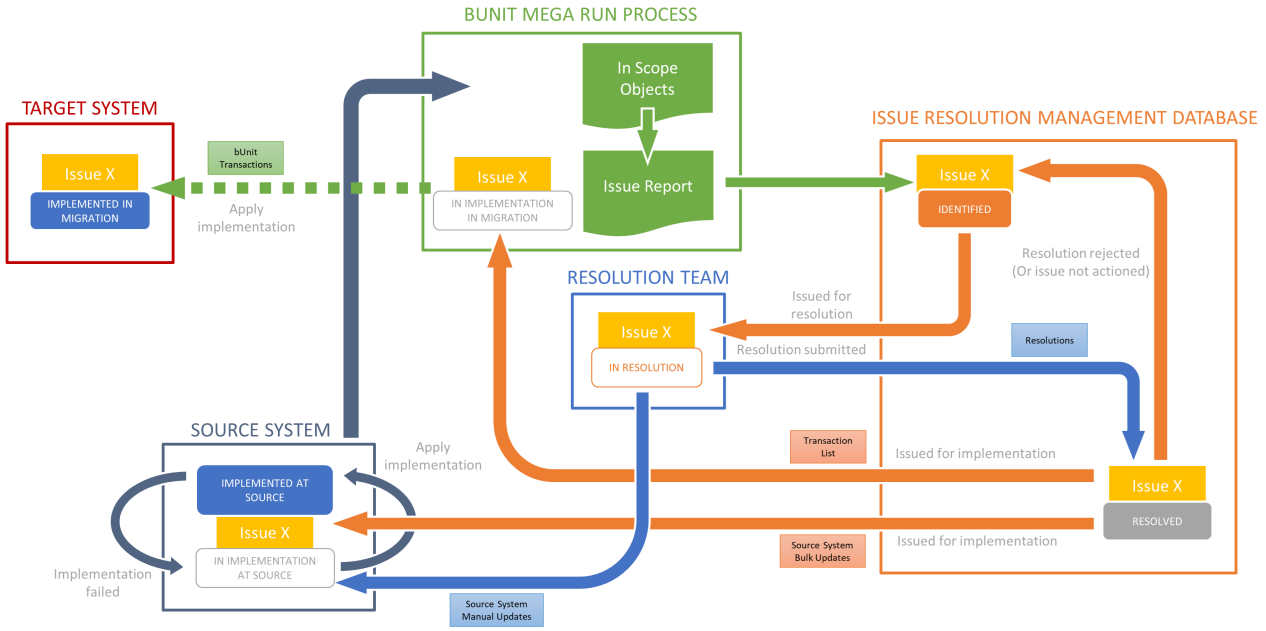
The bCLEARer query architecture will have common general features as well as design specific to the particular process. The common general features include mechanisms for identifying which run a particular query was raised - as well as a means for identifying the equivalence of queries across runs. The design specific to the particular process will involve a classification system for query types relevant to the data used in the process (though a proportion of these will be common - for example, empty cell).

The raison d'etre of raising queries is to have them resolved. We have identified a standard workflow for resolutions - shown graphically below. As this shows the resolution can be implemented inside the migration process or can be at source.



issue resolution workflow

The query management process will run at different levels. There will be an operational resolution process - an example of this is shown graphically below.



example - query resolution management process

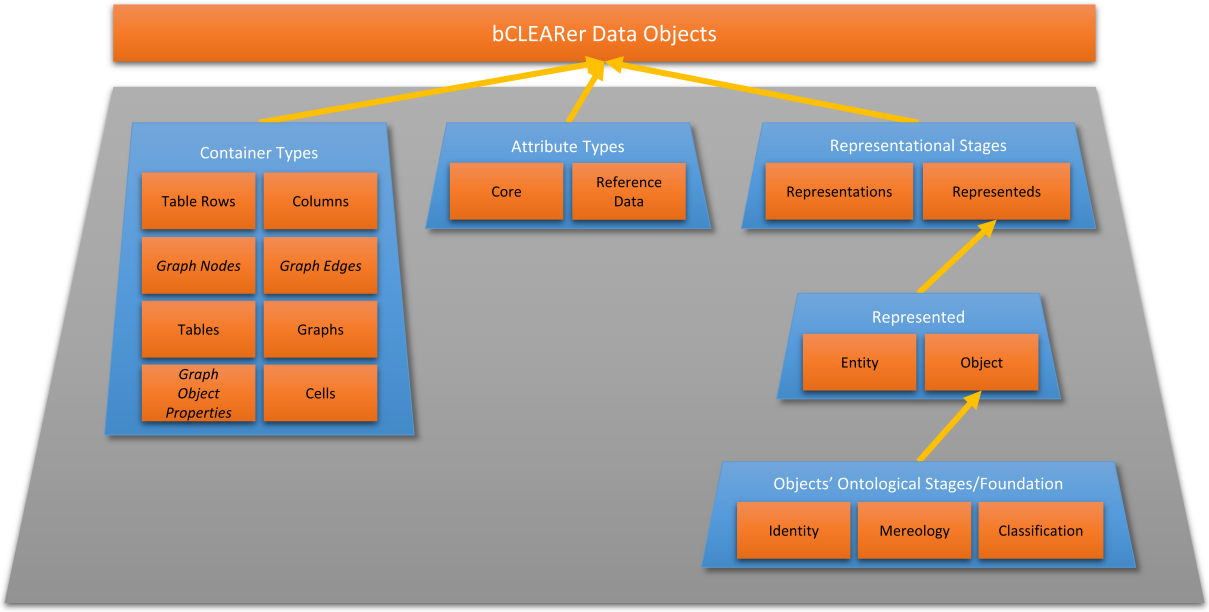
There will also be a dashboard level management - where the volume of queries and the velocity of resolutions will be tracked.

The objects in the query ecosystem have identity - and this has been captured using the BORO Identity Ecosystem (the subject of another eManual). Applying identity in a coherent way, makes the management of queries much easier. It ensures for example, when the same (or equivalent) query appears across runs. This simplifies the tracking of resolutions.

1.6.12.2 bCLEARer Stage Level Pipeline Design

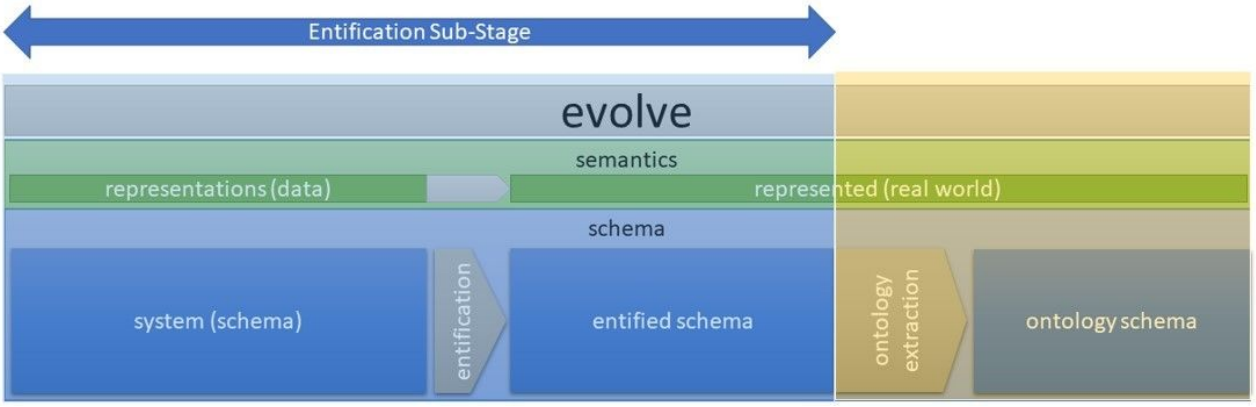
The design of the bCLEARer Stage pipeline needs to be sensitive to the requirements of the specific context. However, we have found that similar patterns emerge over time. So examination or past designs can provide useful insights into current projects.

Based on previous work we have done an initial facet analysis of the data objects that emerge in the bCLEARer process - these are shown graphically below.



bclearer data object facets

The bCLEARer Evolve stage normally requires significant design - and redesign as further requirements emerge. Typically, the early stages of evolve deal with syntactic (data) issues. They then follow a pattern of entity paradigm followed by the BORO ontology paradigm - sometimes with an object paradigm as an intermediate step (described in detail in (Partridge 1996 Business Objects) (see page 140)).



anatomy of the bclearer process - entification substage

1.6.12.3 bCLEARer Universes

Data is transported along the bCLEARer pipeline. Universes are data storage structures that are used to facilitate this transportation. Typically a universe will persist the data from one gate to another - in other words, from the start to the end of a gated pipeline. This is the scope of the universe. The scope has what can be regarded as spatial and temporal aspects. The temporal aspect is the length of the pipeline along which it stores data. The spatial aspect is the range of data in the dataset. This is typically all the data that enters at the start and all new data created along the pipeline. The universe is mutable when in its temporal range, as it needs to store the changes. Outside its temporal range it is immutable - providing an audit inspection source.

Universe scopes are flexible, driven by the specific requirements of the project. For example, a single universe can be created and updated along the whole project, but the most common usage is creating a universe per atomic unit of work (usually a bUnit). This fine grained, there will be situations that require coarser (different) granularity, like bCLEARer sub-stages or domains (see example here[62]).

The design of the universe will follow the design principles, such as Appendix - Aggregated Single Source of Truth Principle (see page 46) and Appendix - Separation of Concerns Principle (see page 75). The latter principle is usually evident in the design of the registries of the universe. These are data collections, usually in the format of a class (example here[63]). Typically, each registry deals with a separate concern.

The network, lattice structure of the overall pipeline means that pipelines can split and merge. Where this happens, there may be a requirement for the universe to split and merge too. A general universe 'algebra' has been designed to accommodate these requirements.

62 https://github.com/boro-alpha/nf_ea_common_tools/blob/master/nf_ea_common_tools_source/b_code/services/general/nf_ea/com/nf_ea_com_universes.py

63 https://github.com/boro-alpha/nf_ea_common_tools/blob/master/nf_ea_common_tools_source/b_code/services/general/nf_ea/com/nf_ea_com_registries.py

1.6.12.4 BORO Clean Coding Principles

Clean coding is a vital tool for the efficiency of bCLEARer code. Over the decades, BORO has developed and documented (internally) its clean coding approach into a working eManual. Their work has drawn inspiration from (Martin 2012 Clean Code) (see page 134) and (Martin 2018 Clean Architecture) (see page 136). This was one topic in the bCLEARer training (Training - bCLEARer - UNICLASS Example (see page 220)). Here is the BORO Clean Coding Quick Style Guide (see page 105) used in that training.

In the longer term, it would make sense for bCLEARer teams to take advantage of BORO's learnings in this area. BORO has a working internal eManual for its clean coding practices. One way to speed up the learning curve would be to develop ways to communicate the BORO documentation inside the organisation.

1.6.12.4.1 BORO Clean Coding Quick Style Guide

This is a version of the BORO Clean Coding Quick Style Guide that is used by coders on bCLEARer projects.

1.6.12.4.1.1 Quick Style Guide

Objects	BORO Standards Applied	Examples
class names	camel case - plural	`class MyObjectTypes:`
class file names	snake case	`my_object_types.py`
constant names	capital case - underscore between words	`IDS_COLUMN_NAME = 'boro_ids'`

all names that are not classes or constants (including folder names)	snake case	`string_variable;`
		`get_my_function_result()`
		
		`\my_process\my_stage_n\my_common_functions\`

string delimiter	single quotes	'example of a BORO standard string in Python'

function names	'action' names	`get_something_from_somewhere();`
		`export_something();`
		`import_my_data()`

Objects	BORO Standards Applied	Examples

file names	‘actor’ names	`data_from_somewhere_getter.py;`
		`data_exporter.py;`
		`data_importer.py`

between instructions	leave an empty line	`my_variable = \`
		`  'my_value'`
		
		`my_second_variable = \`
		`  'my_second_value'`

After colon ':' (function declaration and loops)	do not leave next line empty	`def my_function()`
		`  -> None:`
		`  my_variable = \`
		`  'my_value'`
		
		`  if my_condition:`
		`  my_second_variable = \`
		`  'my_second_value'`

variable assignments	• always in a new line • one space before the equals char (=) and one space after	`my_variable = \` `  get_my_data()`

Objects	BORO Standards Applied	Examples

function arguments (declaration and calls)	• each argument always in a new line
	• in the parameter name - argument value couple in a function call
	  • No spaces around the equals char (=)
	  • Parameter name and argument value in the same line (do not break the line after the equals char (=)
		`def get_my_data(`
		`  first_parameter: str,`
		`  second_parameter:`
		`list):`
		
		`my_data = \`
		`  get_my_data(`
		
		`first_parameter='my string argument',`
		
		`second_parameter=[element_1, element_2, element_3])`

declarating arguments	always specify the type	`def get_my_data(`
		`  first_argument: str,`
		`  second_argument:`
		`list):`

passing parameters	always specify the argument name	`my_data = \`
		`  get_my_data(`
		`  first_argument='my string argument',`
		
		`second_argument=[element_1, element_2, element_3])`

function return type	• always specified in declaration
	• in a new line before the colon of the function
	• if the function doesn't return anything, then it returns **- > None**
		`def get_my_string(`
		`  my_string: str) \`
		`  -> str:`

Objects	BORO Standards Applied	Examples

variable names	meaningful, specific, informative, no acronyms	`x = ‘1234’ - WRONG`
		
		`tmp = ['ID', ‘object_name’, ‘object_parent_ID’] - WRONG`
		
		`number_as_string = ‘1234’`
		
		`temporary_list_of_column_names = ['ID', ‘object_name’, ‘object_parent_ID’]`

private functions	• prefix their names with double underscore (__) • They’re called ‘private’ because they are being used only by the main function in the file, and never been called externally	`def __process_line_of_my_table():`
function design rules	• should only return one item • should only do one thing (separation of concerns) • should be broken down into sub functions as much as possible	

Python files	• should only store one function (unless it’s the case of a suite of functions)
	  • and its private subfunctions
	  • and may be closely related functions
		The hash_creators.py file can hold different functions, such as:
		• create_identity_hasth_string(…)
		• create_tiny_hash_string(…)

Objects	BORO Standards Applied	Examples
comments	• a clean code should not need comments • the code should be clear and self-explanatory • they can be used for development purposes	

'for' and 'if' loops design	• if the for/if loop has more than one statement under it
	  • take out all the loop content into a private function
	• if there are nested for/if loops
	  • they shouldn't be visible
	  • remove them by grouping them into private functions	

'for' loops	'in' block goes in a new line	`for index, row \` `  in table.iterrows():`
hardcoded strings approach	Strings should never be hardcoded. Preferred to: • Define them as constant in separated file/s • Define them as enums in a separate class	
file and folder work	We prefer to use **nf_common** classes **Files** and **Folders**	Nf_Common function library read-only PyCharm requirements.txt line: `git+https://ElBloqueReadOnly:7hKHi78Qv4bw@bitbucket.org/nf_py_g6_team/nf_common.git`

Objects	BORO Standards Applied	Examples

file/folder path work	To ensure paths can work for different platforms, the use of **os.path.join()** (and related functions) is encouraged, along with the OS separator character **os.sep**, rather than pure string manipulation	`PARENT_FOLDER_PATH = \`
		`  Path(`
		`  'C:',`
		`  os.sep,`
		`  'RootDirectory',`
		`  'Subdirectory',`
		`  'Storage')`

general scope functions	When there is the need for a general function, it is worth a check to the **nf_common** function library. If we can use something from the nf_common function library, we use it and we save writing new redundant code	Nf_Common function library read-only PyCharm requirements.txt line: `git+https://ElBloqueReadOnly:7hKHi78Qv4bw@bitbucket.org/nf_py_g6_team/nf_common.git`

use of 'orchestrators'	When a function is managing a sequence or group of different proceses we tend to call it an 'orchestrator' function.
	• orchestrators can be nested (see example)
		file: **standard_analysis_process_orchestrator**.py
		typical function name and structure:
		`def orchestrate_standard_analysis_process(…):`
		`  import_data(…)`
		`  analyse_data(…)`
		
		`orchestrate_export_analysis(…)`

1.6.12.5 BORO Modelling Resources

There are significant resources for BORO Modelling. The prime resource is the book (Partridge 1996 Business Objects) (see page 140). BORO provides access to articles and presentations on BORO is its library - (BORO Solutions Web Library) (see page 200). Many of these have been authored by Chris Partridge and are available on his https://www.academia.edu/ profile page - (Partridge - Academia profile) (see page 201).

1.6.12.6 Gate Object Accounting

Within bCLEARer, the deployment of a gate architecture (see Nested gated pipeline architecture (see page 15)) creates an opportunity for object accounting, in other words, the use of counts and hashes to check whether the amount of objects is consistent with the objectives of the process - and consistent between reruns of the process. Our experience is that this accounting is often essential for the quality control and reliability of the process and, as such, it has been incorporated into the bCLEARer process. Given the scale of the process, this usually also creates a requirement for the accounting to be automated.

For each bCLEARer process, the specific features of its overall accounting architecture need to be designed. Firstly, there is a requirement to design the 'balance sheet' architecture for each gate - the statement of the types of object that exist at that gate. Secondly, there is a requirement to design the 'profit and loss' process architecture for each gate - a statement of how the object universe has changed (how the types of objects have changed - what types of changes and by how much) between the gates.

The 'balance sheet' metaphor is useful as it captures the notion of the description of a process at a point in time (the gate snapshot).

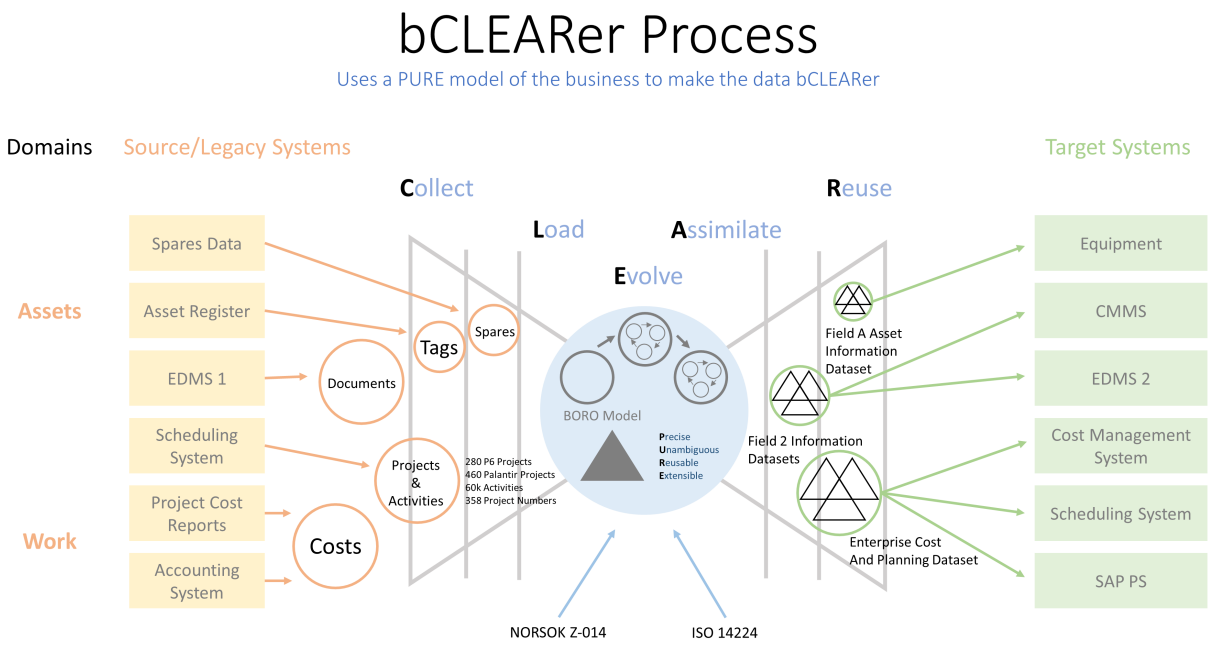
While the ‘accounting’ metaphor works well for the 'balance sheet' it is a bit stretched for the 'profit and loss' process. This is because there is a larger variety of ways in which objects can be transformed in the bCLEARer process than in typical financial accounting. The bCLEARer accounting design for the 'profit and loss' process architecture worked with the notion of accounting event. At the simplest level, the event would be a simple copy. In this case, the checking algorithm would check that the counts/hashes are equal. Other simple events are merges and splits of tables - these would have their simple corresponding checking algorithms. One of the challenges is developing a suite of checking algorithms that covers the full range of the transformation.

There is an interplay between the pipeline architecture and the accounting architecture. One can significantly simplify the checking algorithms in the accounting architecture through designs in the pipeline architecture. For example, merges and splits are simpler to account for, so it helps if these are used in the pipeline design.

In the evolution of bCLEARer gate object accounting, the earliest implementations worked mainly with counts. Over time, many of their functions have been replaced with hashes. Significant work has been done to develop hashes that work well with checking algorithms - some of this work is embedded in the BORO Identity Ecosystem (bie) the subject of a subsequent manual.

1.6.12.7 Multi-System bCLEARer Architecture

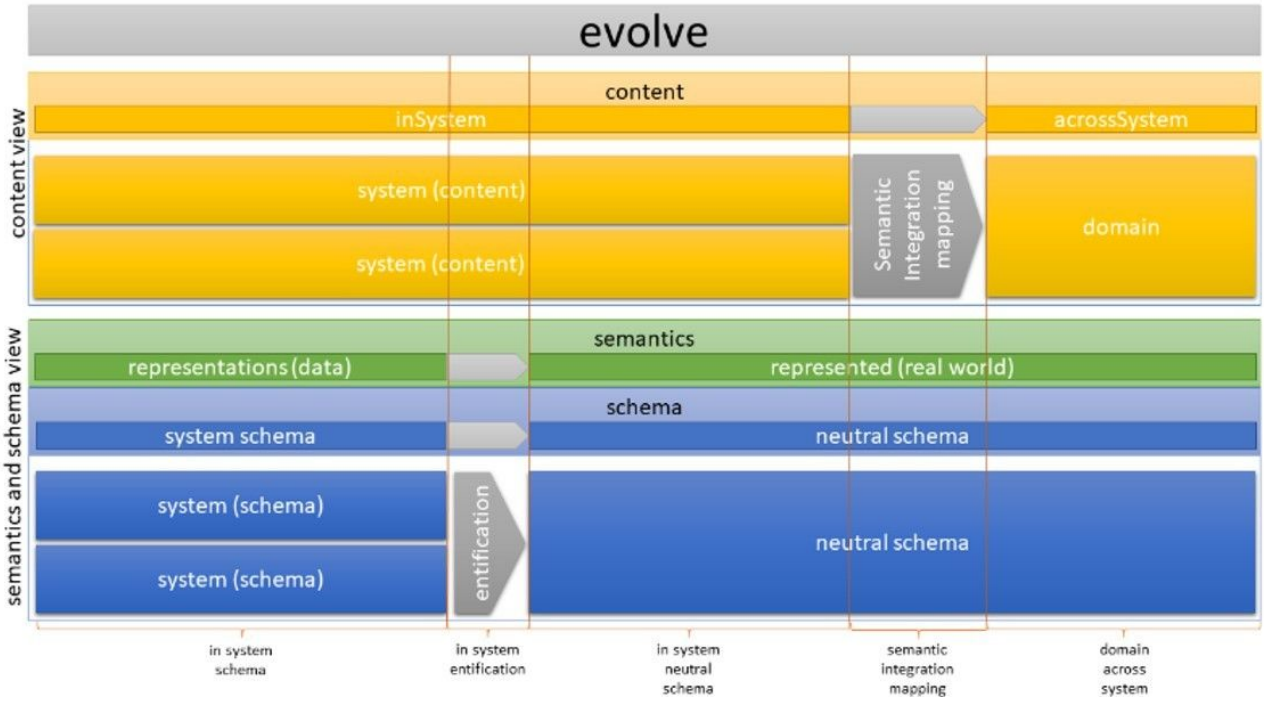
bCLEARer processes typically involve a number of datasets from different sources (systems) - as shown in the graphic below. Indeed it is recommended that more then one system is used at source as this provides a basis for triangulation across the systems which leads to more accurate models.



bclearer process - multi-system

As was noted in bCLEARer Stage Level Pipeline Design (see page 102) "The design of the bCLEAR Stage pipeline needs to be sensitive to the requirements of the specific context." The use of multiple systems is a particularly relevant feature of the context for pipeline design.

Principles, such as Appendix - Separation of Concerns Principle (see page 75) and Appendix - Single-Transformation (Responsibility) Principle (STP) (see page 78), suggest a natural architectural style where the schema or language in which the data is held is separated from the content and these concerns are handled separately. This leads us to designs where the systems are initially processed independently (as separate concerns) and then together. These are colloquially known as the '*in-system*' and '*across-system*' phases of the process. The anatomy for a typical multi-system bCLEARer thin slice process is shown in figure below. This will typically be reflected in the pipeline architecture.



anatomy of a typical multi system bclearer process

1.7 eManuals bibliography

The following eManuals are referred to in this eManual:

- BORO, BORO Foundation and bCLEARer --- A very brief introduction (see page 116)

1.7.1 BORO, BORO Foundation and bCLEARer --- A very brief introduction

BORO. 2024. 'BORO, BORO Foundation and bCLEARer: A very brief introduction'. (see page 116)

1.8 Bibliography

- (Arjoon 2006 Striking a Balance Between Rules and Principles-Based Approaches for Effective Governance) (see page 120) — Arjoon, Surendra. 2006. 'Striking a Balance Between Rules and Principles-Based Approaches for Effective Governance: A Risks-Based Approach'. Journal of Business Ethics 68(1):53–82. doi: 10.1007/s10551-006-9040-6 https://link.springer.com/article/10.1007/s10551-006-9040-6.
- (Bass 2012 Software Architecture in Practice) (see page 121) — Bass, Len, Paul Clements, and Kazman. Software Architecture in Practice. 3rd ed. Addison-Wesley Professional, 2012.
- (Bennett 2006 Rules, Principles and Judgments in Accounting Standards) (see page 125) — Bennett, Bruce, Michael Bradbury, and Helen Prangnell. 2006. 'Rules, Principles and Judgments in Accounting Standards'. Abacus 42(2):189–204. doi: 10.1111/j.1467-6281.2006.00197.x https://onlinelibrary.wiley.com/doi/10.1111/j.1467-6281.2006.00197.x.
- (Benston 2006 Principles versus Rules-based Accounting Standards) (see page 126) — Benston, George J., Michael Bromwich, and Alfred Wagenhofer. 2006. 'Principles versus Rules-based Accounting Standards: The FASB's Standard Setting Strategy'. Abacus 42(2):165–88. doi: 10.1111/j.1467-6281.2006.00196.x https://onlinelibrary.wiley.com/doi/10.1111/j.1467-6281.2006.00196.x.
- (Braithwaite 2002 Rules and Principles) (see page 127) — Braithwaite, John Bradford. 2002. 'Rules and Principles: A Theory of Legal Certainty'. SSRN Electronic Journal. doi: 10.2139/ssrn.329400 https://papers.ssrn.com/sol3/papers.cfm?abstract_id=329400.
- (de Cesare 2016 BORO as a Foundation to Enterprise Ontology) (see page 128) — de Cesare, Sergio, and Chris Partridge. "BORO as a Foundation to Enterprise Ontology." Journal of Information Systems 30, no. 2 (Summer 2016) (2016): 83–112. https://doi.org/10.2308/isys-51428 https://doi.org/10.2308/isys-51428
- (DeMarco 1979 Structured Analysis and System Specification) (see page 129) — DeMarco, Tom. 1979. Structured Analysis and System Specification. Englewood Cliffs, N.J: Prentice-Hall.
- (Dijkstra 1974 On the role of scientific thought) (see page 130) — Dijkstra, Edsger (1974). "On the role of scientific thought". E.W. Dijkstra Archive, Center for American History, University of Texas at Austin https://en.wikipedia.org/wiki/University_of_Texas_at_Austin. EWD447.
- (Hilf 2001 Power, Rules and Principles) (see page 131) — Hilf, M. 2001. 'Power, Rules and Principles - Which Orientation for WTO/GATT Law?' Journal of International Economic Law 4(1):111–30. doi: 10.1093/jiel/4.1.111 https://doi.org/10.1093/jiel/4.1.111.
- (Martin 2003 Agile Software Development) (see page 132) — Robert C. Martin. 2003. Agile Software Development: Principles, Patterns, and Practices. Upper Saddle River, NJ: Prentice Hall/Pearson Education.
- (Martin 2005 The Principles of OOD) (see page 133) — Martin, R. C. (2005). The Principles of OOD. The Principles of OOD. http://butunclebob.com/ http://butunclebob.com/
- (Martin 2012 Clean Code) (see page 134) — Martin, Robert C. 2012. Clean Code: A Handbook of Agile Software Craftsmanship. Repr. Upper Saddle River, NJ Munich: Prentice Hall.
- (Martin 2014 The Single Responsibility Principle) (see page 135) — Martin, Robert C. (2014). "The Single Responsibility Principle https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html
- (Martin 2018 Clean Architecture) (see page 136) — Martin, Robert C. 2018. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston: Prentice Hall.
- (Nelson 2003 Rules, Behavioral Evidence on the Effects of Principles- and Rules-Based Standards) (see page 137) — Nelson, Mark W. 2003. 'Behavioral Evidence on the Effects of Principles- and Rules-Based Standards'.

SSRN Electronic Journal. doi: 10.2139/ssrn.360441 https://papers.ssrn.com/sol3/papers.cfm?abstract_id=360441.

- (Page-Jones 1988 The Practical Guide to Structured Systems Design) (see page 138) — Page-Jones, Meilir (1988). The Practical Guide to Structured Systems Design. Yourdon Press Computing Series.
- (Parnas 1972 Decomposing Systems into Modules) (see page 139) — Parnas, David Lorge. "On the Criteria to Be Used in Decomposing Systems into Modules." Communications of the ACM 15, no. 12 (1972): 1053–58.
- (Partridge 1996 Business Objects) (see page 140) — Partridge, Chris. Business Objects: Re-Engineering for Re-Use. 1st Edition. Oxford: Butterworth-Heinemann, 1996.
- (Partridge 2016 BORO Foundational Ontology's Meta-ontological Choices) (see page 141) — Partridge, C. (2016, June 6). BORO Foundational Ontology's Meta-ontological Choices. Onto.Com, Co-Located with FOIS 2016.
- (Partridge 2016 Grounding for Ontological Architecture Quality) (see page 142) — Partridge, C., & Cesare, S. de. (2016). Grounding for Ontological Architecture Quality: Metaphysical Choices. In S. Link & J. C. Trujillo (Eds.), Advances in Conceptual Modeling: ER 2016 Workshops, AHA, MoBiD, MORE-BI, MReBA, QMMQ, SCME, and WM2SP, Gifu, Japan, November 14–17, 2016, Proceedings (Vol. 9975, pp. 9-15–XXIII, 251). Springer International Publishing. 10.1007/978-3-319-47717-6 https://link.springer.com/book/10.1007/978-3-319-47717-6
- (Partridge 2017 Developing an Ontological Sandbox) (see page 143) — Partridge, C., deCesare, S., Mitchell, A., Gailly, F., & Khan, M. (2017). Developing an Ontological Sandbox: Investigating Multi-Level Modelling's Possible Metaphysical Structures. Proceedings of MODELS 2017 Satellite Event: Workshops (ModComp, ME, EXE, COMMitMDE, MRT, MULTI, GEMOC, MoDeVVa, MDETools, FlexMDE, MDEbug), Posters, Doctoral Symposium, Educator Symposium, ACM Student Research Competition, and Tools and Demonstrations Co-Located with ACM/IEEE 20th International Conference on Model Dri
- (Partridge 2019 Coordinate Systems) (see page 144) — Partridge, C., Mitchell, A., Loneragan, M., Atkinson, H., de Cesare, S., & Khan, M. (2019). Coordinate Systems: Level Ascending Ontological Options. 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), 78–87. https://www.academia.edu/40354620 https://www.academia.edu/40354620
- (Partridge 2020 A Framework for Composition) (see page 145) — Partridge, C., Mitchell, A., & Grenon, P. (2021). A Framework for Composition: A Step Towards a Foundation for Assembly. CDBB. https://doi.org/10.17863/CAM.66459 https://doi.org/10.17863/CAM.66459
- (Partridge 2020 A Survey of Top-Level Ontologies) (see page 146) — Partridge, C., Mitchell, A., Cook, A., Sullivan, J., & West, M. (2020). A Survey of Top-Level Ontologies - to inform the ontological choices for a Foundation Data Model. CDBB. https://doi.org/10.17863/CAM.58311 https://doi.org/10.17863/CAM.58311
- (Partridge 2020 Implicit Requirements for Ontological Multi-Level Types in the UNICLASS Classification) (see page 147) — Partridge, C., Mitchell, A., Da Silva, M., Soto, O. X., West, M., Khan, M., & De Cesare, S. (2020). Implicit requirements for ontological multi-level types in the UNICLASS classification. Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, 1–8. https://doi.org/10.1145/3417990.3421414 https://doi.org/10.1145/3417990.3421414
- (Partridge 2020 The Fantastic Combinations and Permutations of Co-ordinate Systems Characterising Options) (see page 148) — Partridge, C., Mitchell, A., Loneragan, M., Atkinson, H., de Cesare, S., & Khan, M. (2020). The Fantastic Combinations and Permutations of Co-ordinate Systems' Characterising Options: The Game of Constructional Ontology.
- (Reade 1989 Elements of Functional Programming) (see page 149) — Reade, Chris. 1989. Elements of Functional Programming. Wokingham: Addison-Wesley.

- (Ritchie 1984 The Evolution of the Unix Time-Sharing System) (see page 150) — Ritchie, Dennis. “The Evolution of the Unix Time-Sharing System.” AT&T Bell Laboratories Technical Journal 63, no. 6 Part 2 (1984): 1577–93. https://ieeexplore.ieee.org/document/6771908 https://ieeexplore.ieee.org/document/6771908
- (Schlag 1985 Rules and Standards) (see page 151) — Pierre Schlag, Rules and Standards, 33 UCLA L. Rev. 379 (1985), available at https://scholar.law.colorado.edu/faculty-articles/1034 https://scholar.law.colorado.edu/faculty-articles/1034.
- (Stevens 1974 Structured Design) (see page 152) — Stevens, W. P., G. J. Myers, and L. L. Constantine. 1974. 'Structured Design'. IBM Systems Journal 13(2):115–39. doi: 10.1147/sj.132.0115 https://ieeexplore.ieee.org/document/5388187.
- (West 2006 Developing an Ontology-Based Framework for the Shell Downstream Business) (see page 153) — West, M., Partridge, C., & Lycett, M. (2006, December 14). Enterprise Data Modelling: Developing an Ontology-Based Framework for the Shell Downstream Business. Second Workshop on Formal Ontologies Meet Industry, FOMI 2006.

1.8.1 (Arjoon 2006 Striking a Balance Between Rules and Principles-Based Approaches for Effective Governance)

Arjoon, Surendra. 2006. 'Striking a Balance Between Rules and Principles-Based Approaches for Effective Governance: A Risks-Based Approach'. *Journal of Business Ethics* 68(1):53–82. doi: 10.1007/s10551-006-9040-6[64]. (see page 120)

64 https://link.springer.com/article/10.1007/s10551-006-9040-6

1.8.2 (Bass 2012 Software Architecture in Practice)

Bass, Len, Paul Clements, and Kazman. *Software Architecture in Practice*. 3rd ed. Addison-Wesley Professional, 2012. (see page 121)

p. 239 - 240

The standard pipe and filter pattern

Pipe-and-Filter Pattern

Context: Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.

Problem: Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.

Solution: The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

> *Data transformation systems are typically structured as pipes and filters, with each filter responsible for one part of the overall transformation of the input data. The independent processing at each step supports reuse, parallelization, and simplified reasoning about overall behaviour. Often such systems constitute the front end of signal-processing applications. These systems receive sensor data at a set of initial filters; each of these filters compresses the data and performs initial processing (such as smoothing). Downstream filters reduce the data further and do synthesis across data derived from different sensors. The final filter typically passes its data to an application, for example providing input to modelling or visualization tools*

Key terms

terms	descriptions
Overview	Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.

terms	descriptions
Elements	*Filter*, which is a component that transforms data read on its input port(s) to data written on its output port(s). Filters can execute concurrently with each other. Filters can incrementally transform data; that is, they can start producing output as soon as they start processing input. Important characteristics include processing rates, input/output data formats, and the transformation executed by the filter. *Pipe*, which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through. Important characteristics include buffer size, protocol of interaction, transmission speed, and format of the data that passes through a pipe.
Relations	The *attachment* relation associates the output of filters with the input of pipes and vice versa.
Constraints	Pipes connect filter output ports to filter input ports. Connected filters must agree on the type of data being passed along the connecting pipe. Specializations of the pattern may restrict the association of components to an acyclic graph or a linear sequence, sometimes called a pipeline. Other specializations may prescribe that components have certain named ports, such as the *stdln*, *stdout*, and *stderr* ports of UNIX filters.
Weaknesses	The pipe-and-filter pattern is typically not a good choice for an interactive system. Having large numbers of independent filters can add substantial amounts of computational overhead. Pipe-and-filter systems may not be appropriate for long-running computations.

(see page 123)

1.8.2.1 The standard pipe and filter pattern

1.8.2.1.1 13.1. Architectural Patterns

p. 239-240

1.8.2.1.1.1 Pipe-and-Filter Pattern

Context: Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.

Problem: Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.

Solution: The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

> *Data transformation systems are typically structured as pipes and filters, with each filter responsible for one part of the overall transformation of the input data. The independent processing at each step supports reuse, parallelization, and simplified reasoning about overall behaviour. Often such systems constitute the front end of signal-processing applications. These systems receive sensor data at a set of initial filters; each of these filters compresses the data and performs initial processing (such as smoothing). Downstream filters reduce the data further and do synthesis across data derived from different sensors. The final filter typically passes its data to an application, for example providing input to modelling or visualization tools*

1.8.2.1.1.2 Key terms

terms	descriptions
Overview	Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.
Elements	*Filter*, which is a component that transforms data read on its input port(s) to data written on its output port(s). Filters can execute concurrently with each other. Filters can incrementally transform data; that is, they can start producing output as soon as they start processing input. Important characteristics include processing rates, input/output data formats, and the transformation executed by the filter. *Pipe*, which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through. Important characteristics include buffer size, protocol of interaction, transmission speed, and format of the data that passes through a pipe.

terms	descriptions
Relations	The *attachment* relation associates the output of filters with the input of pipes and vice versa.
Constraints	Pipes connect filter output ports to filter input ports. Connected filters must agree on the type of data being passed along the connecting pipe. Specializations of the pattern may restrict the association of components to an acyclic graph or a linear sequence, sometimes called a pipeline. Other specializations may prescribe that components have certain named ports, such as the *stdln*, *stdout*, and *stderr* ports of UNIX filters.
Weaknesses	The pipe-and-filter pattern is typically not a good choice for an interactive system. Having large numbers of independent filters can add substantial amounts of computational overhead. Pipe-and-filter systems may not be appropriate for long-running computations.

(see page 123)

(Bass 2012 Software Architecture in Practice)

Bass, Len, Paul Clements, and Kazman. *Software Architecture in Practice*. 3rd ed. Addison-Wesley Professional, 2012. (see page 121)

1.8.3 (Bennett 2006 Rules, Principles and Judgments in Accounting Standards)

Bennett, Bruce, Michael Bradbury, and Helen Prangnell. 2006. 'Rules, Principles and Judgments in Accounting Standards'. *Abacus* 42(2):189–204. doi: 10.1111/j.1467-6281.2006.00197.x[65]. (see page 125)

65 https://onlinelibrary.wiley.com/doi/10.1111/j.1467-6281.2006.00197.x

1.8.4 (Benston 2006 Principles versus Rules-based Accounting Standards)

Benston, George J., Michael Bromwich, and Alfred Wagenhofer. 2006. 'Principles versus Rules-based Accounting Standards: The FASB's Standard Setting Strategy'. *Abacus* 42(2):165–88. doi: 10.1111/j.1467-6281.2006.00196.x[66]. (see page 126)

66 https://onlinelibrary.wiley.com/doi/10.1111/j.1467-6281.2006.00196.x

1.8.5 (Braithwaite 2002 Rules and Principles)

Braithwaite, John Bradford. 2002. 'Rules and Principles: A Theory of Legal Certainty'. *SSRN Electronic Journal*. doi: 10.2139/ssrn.329400[67]. (see page 127)

67 https://papers.ssrn.com/sol3/papers.cfm?abstract_id=329400

1.8.6 (de Cesare 2016 BORO as a Foundation to Enterprise Ontology)

de Cesare, Sergio, and Chris Partridge. “BORO as a Foundation to Enterprise Ontology.” *Journal of Information Systems* 30, no. 2 (Summer 2016) (2016): 83–112. https://doi.org/10.2308/isys-51428 (see page 128)

1.8.7 (DeMarco 1979 Structured Analysis and System Specification)

DeMarco, Tom. 1979. *Structured Analysis and System Specification*. Englewood Cliffs, N.J: Prentice-Hall.
(see page 129)

1.8.8 (Dijkstra 1974 On the role of scientific thought)

Dijkstra, Edsger (1974). "On the role of scientific thought". E.W. Dijkstra Archive, Center for American History, University of Texas at Austin[68]. EWD447. (see page 130)

68 https://en.wikipedia.org/wiki/University_of_Texas_at_Austin

1.8.9 (Hilf 2001 Power, Rules and Principles)

Hilf, M. 2001. 'Power, Rules and Principles - Which Orientation for WTO/GATT Law?' *Journal of International Economic Law* 4(1):111–30. doi: 10.1093/jiel/4.1.111[69]. (see page 131)

69 https://doi.org/10.1093/jiel/4.1.111

1.8.10 (Martin 2003 Agile Software Development)

Robert C. Martin. 2003. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice Hall/Pearson Education. (see page 132)

1.8.11 (Martin 2005 The Principles of OOD)

Martin, R. C. (2005). The Principles of OOD. *The Principles of OOD*. http://butunclebob.com/ (see page 133)

1.8.12 (Martin 2012 Clean Code)

Martin, Robert C. 2012. *Clean Code: A Handbook of Agile Software Craftsmanship*. Repr. Upper Saddle River, NJ Munich: Prentice Hall. (see page 134)

1.8.13 (Martin 2014 The Single Responsibility Principle)

Martin, Robert C. (2014). "The Single Responsibility Principle https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html (see page 135)

1.8.14 (Martin 2018 Clean Architecture)

Martin, Robert C. 2018. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston: Prentice Hall. (see page 136)

1.8.15 (Nelson 2003 Rules, Behavioral Evidence on the Effects of Principles- and Rules-Based Standards)

Nelson, Mark W. 2003. 'Behavioral Evidence on the Effects of Principles- and Rules-Based Standards'. *SSRN Electronic Journal*. doi: 10.2139/ssrn.360441[70]. (see page 137)

70 https://papers.ssrn.com/sol3/papers.cfm?abstract_id=360441

1.8.16 (Page-Jones 1988 The Practical Guide to Structured Systems Design)

Page-Jones, Meilir (1988). The Practical Guide to Structured Systems Design. Yourdon Press Computing Series. (see page 138)

1.8.17 (Parnas 1972 Decomposing Systems into Modules)

Parnas, David Lorge. “On the Criteria to Be Used in Decomposing Systems into Modules.” *Communications of the ACM* 15, no. 12 (1972): 1053–58. (see page 139)

1.8.18 (Partridge 1996 Business Objects)

Partridge, Chris. *Business Objects: Re-Engineering for Re-Use*. 1st Edition. Oxford: Butterworth-Heinemann, 1996. (see page 140)

1.8.19 (Partridge 2016 BORO Foundational Ontology's Meta-ontological Choices)

Partridge, C. (2016, June 6). BORO Foundational Ontology’s Meta-ontological Choices. Onto.Com, Co-Located with FOIS 2016. (see page 141)

1.8.20 (Partridge 2016 Grounding for Ontological Architecture Quality)

Partridge, C., & Cesare, S. de. (2016). Grounding for Ontological Architecture Quality: Metaphysical Choices. In S. Link & J. C. Trujillo (Eds.), Advances in Conceptual Modeling: ER 2016 Workshops, AHA, MoBiD, MORE-BI, MReBA, QMMQ, SCME, and WM2SP, Gifu, Japan, November 14–17, 2016, Proceedings (Vol. 9975, pp. 9-15–XXIII, 251). Springer International Publishing. 10.1007/978-3-319-47717-6[71] (see page 142)

71 https://link.springer.com/book/10.1007/978-3-319-47717-6

1.8.21 (Partridge 2017 Developing an Ontological Sandbox)

Partridge, C., deCesare, S., Mitchell, A., Gailly, F., & Khan, M. (2017). Developing an Ontological Sandbox: Investigating Multi-Level Modelling's Possible Metaphysical Structures. *Proceedings of MODELS 2017 Satellite Event: Workshops (ModComp, ME, EXE, COMMitMDE, MRT, MULTI, GEMOC, MoDeVVa, MDETools, FlexMDE, MDEbug), Posters, Doctoral Symposium, Educator Symposium, ACM Student Research Competition, and Tools and Demonstrations Co-Located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017)*, *2019*, 226–234. http://ceur-ws.org/Vol-2019/multi_3.pdf (see page 143)

1.8.22 (Partridge 2019 Coordinate Systems)

Partridge, C., Mitchell, A., Loneragan, M., Atkinson, H., de Cesare, S., & Khan, M. (2019). Coordinate Systems: Level Ascending Ontological Options. *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 78–87. https://www.academia.edu/40354620
(see page 144)

1.8.23 (Partridge 2020 A Framework for Composition)

Partridge, C., Mitchell, A., & Grenon, P. (2021). *A Framework for Composition: A Step Towards a Foundation for Assembly*. CDBB. https://doi.org/10.17863/CAM.66459 (see page 145)

1.8.24 (Partridge 2020 A Survey of Top-Level Ontologies)

Partridge, C., Mitchell, A., Cook, A., Sullivan, J., & West, M. (2020). *A Survey of Top-Level Ontologies - to inform the ontological choices for a Foundation Data Model*. CDBB. https://doi.org/10.17863/CAM.58311 (see page 146)

1.8.25 (Partridge 2020 Implicit Requirements for Ontological Multi-Level Types in the UNICLASS Classification)

Partridge, C., Mitchell, A., Da Silva, M., Soto, O. X., West, M., Khan, M., & De Cesare, S. (2020). Implicit requirements for ontological multi-level types in the UNICLASS classification. *Proceedings of the 23rd ACM/ IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 1–8. https://doi.org/10.1145/3417990.3421414 (see page 147)

1.8.26 (Partridge 2020 The Fantastic Combinations and Permutations of Co-ordinate Systems Characterising Options)

Partridge, C., Mitchell, A., Loneragan, M., Atkinson, H., de Cesare, S., & Khan, M. (2020). *The Fantastic Combinations and Permutations of Co-ordinate Systems' Characterising Options: The Game of Constructional Ontology*. (see page 148)

1.8.27 (Reade 1989 Elements of Functional Programming)

Reade, Chris. 1989. *Elements of Functional Programming*. Wokingham: Addison-Wesley. (see page 149)

1.8.28 (Ritchie 1984 The Evolution of the Unix Time-Sharing System)

Ritchie, Dennis. “The Evolution of the Unix Time-Sharing System.” *AT&T Bell Laboratories Technical Journal* 63, no. 6 Part 2 (1984): 1577–93. https://ieeexplore.ieee.org/document/6771908 (see page 150)

1.8.29 (Schlag 1985 Rules and Standards)

Pierre Schlag, *Rules and Standards*, 33 UCLA L. Rev. 379 (1985), *available at* https://scholar.law.colorado.edu/faculty-articles/1034. (see page 151)

1.8.30 (Stevens 1974 Structured Design)

Stevens, W. P., G. J. Myers, and L. L. Constantine. 1974. 'Structured Design'. *IBM Systems Journal* 13(2):115–39. doi: 10.1147/sj.132.0115[72]. (see page 152)

72 https://ieeexplore.ieee.org/document/5388187

1.8.31 (West 2006 Developing an Ontology-Based Framework for the Shell Downstream Business)

West, M., Partridge, C., & Lycett, M. (2006, December 14). Enterprise Data Modelling: Developing an Ontology-Based Framework for the Shell Downstream Business. Second Workshop on Formal Ontologies Meet Industry, FOMI 2006. (see page 153)

1.9 Acknowledgements

Lead Authors:

Chris Partridge[73]

Anne Guinard[74]

Andy Mitchell[75]

Contributors:

Mesbah A. Khan[77]

Authors:

Oscar Xiberta Soto[76]

Usage

We've made this eManual available for use on bCLEARer® projects. It's offered as-is. We hope it helps on your projects, but you should recognise that it might not work perfectly or suit your needs. If issues arise—like damages or disputes—we're not responsible, no matter how you use it.

BORO Solutions can be contacted through https://borosolutions.net/contact

73 https://www.linkedin.com/in/partridgechris
74 https://www.linkedin.com/in/anne-guinard-62323647/
75 https://www.linkedin.com/in/andrew-mitchell-1285a520
76 https://www.linkedin.com/in/oscar-xiberta-soto-a0829746
77 https://www.linkedin.com/in/mesbahkhan/

2 Resources

2.1 Resources - Additional Pages

2.1.1 About

bCLEARer stands at the forefront of digital transformation, championing an evolutionary approach to harnessing digitization and digitalization opportunities. It guides information on a transformative journey, curating its evolution into fitter forms, ones more suited for computing, that deliver increased value.

To accomplish this, bCLEARer has evolved an architecture framework for semantic data pipelines, along with a methodology for engineering these pipelines. This manual explains the framework.

2.1.2 Copyright

2.1.2.1 Copyright BORO Solutions

 (see page 160)

2.1.3 Version 1 - Initial

This initial version of the eManual (version 1) is a based upon internal ‘living’ documentation developed for a couple of projects. The content has been minimally rewritten to fit into the current eManual structure. It is intended to contain useful information that has been collated as part of the maintenance of a ‘living working document’. It is not intended to be an authoritative or comprehensive overview of the topic.

It is planned, if this version proves useful, to continue to extend the eManual making it more authoritative and comprehensive. (see page 161)

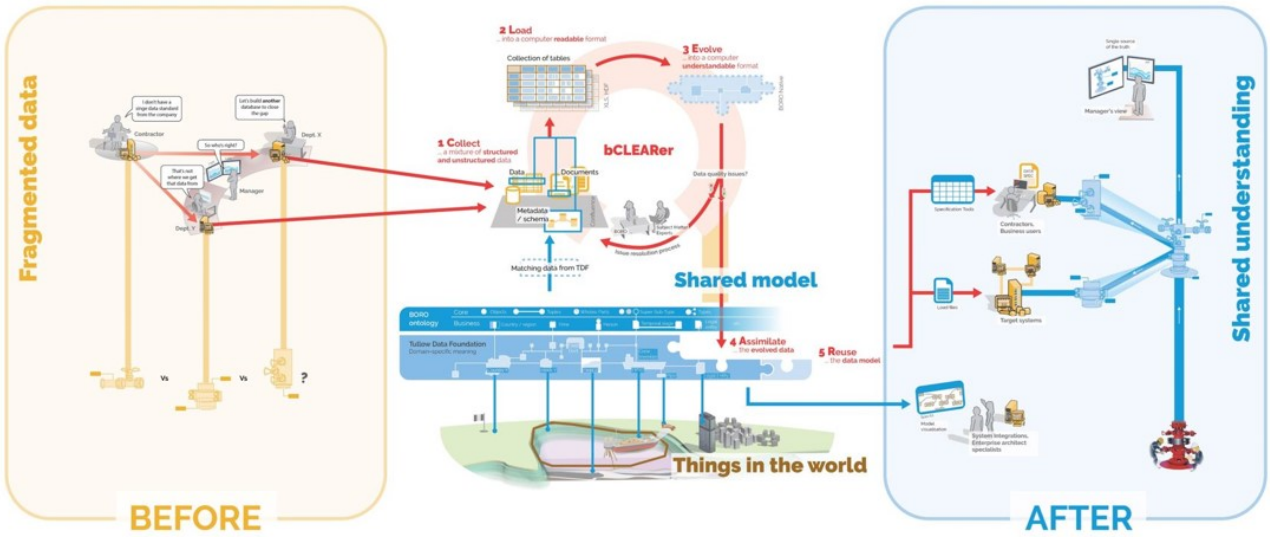
2.1.4 Usage

We've made this eManual available for use on bCLEARer® projects. It's offered as-is. We hope it helps on your projects, but you should recognise that it might not work perfectly or suit your needs. If issues arise—like damages or disputes—we're not responsible, no matter how you use it.

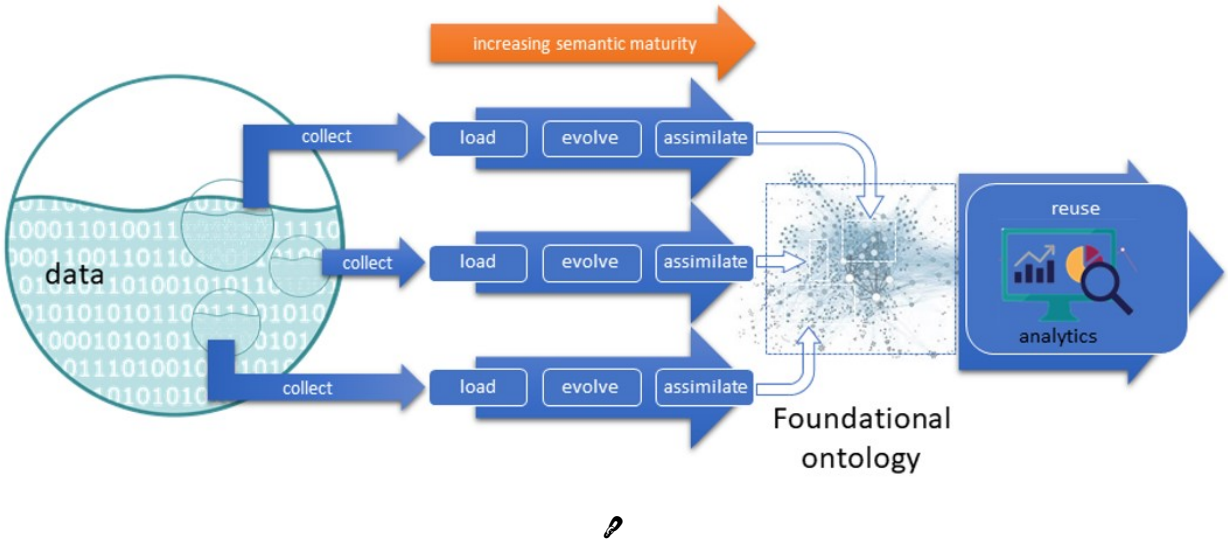
BORO Solutions can be contacted through https://borosolutions.net/contact

2.2 Resources - Graphics

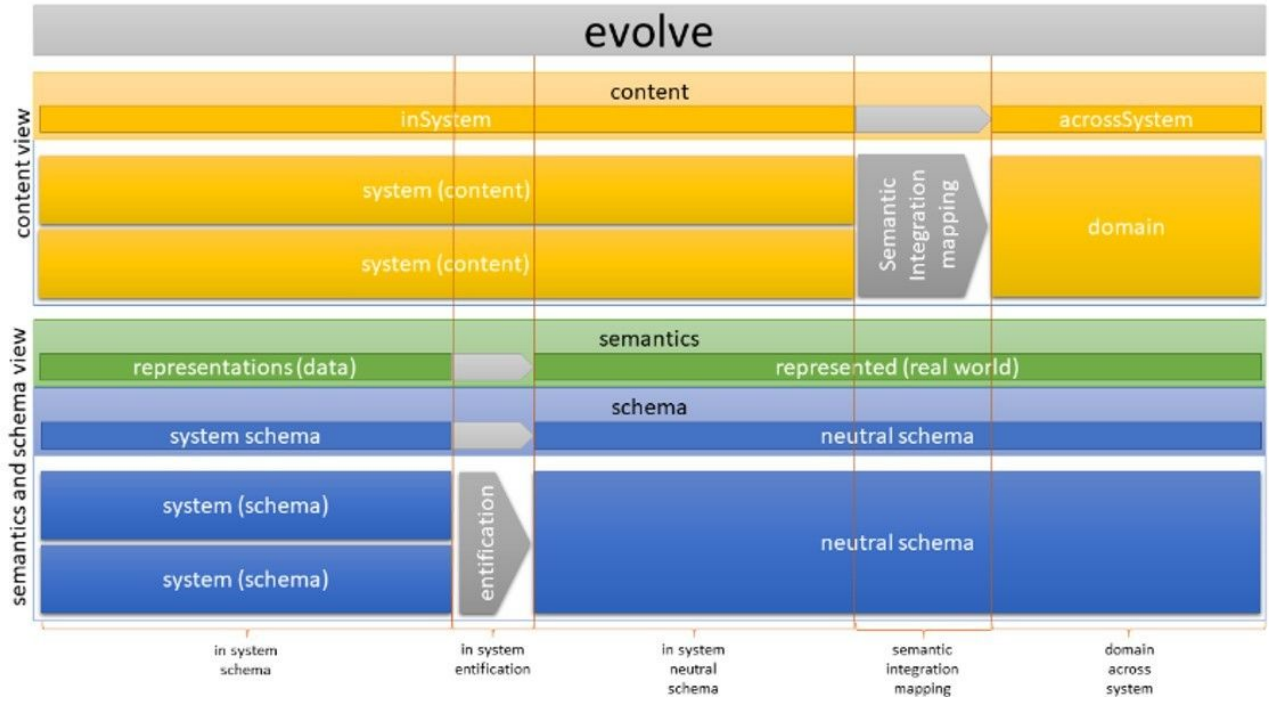
2.2.1 a graphical view of the iterative bclearer approach



2.2.2 a schematic view of the iterative bclearer approach

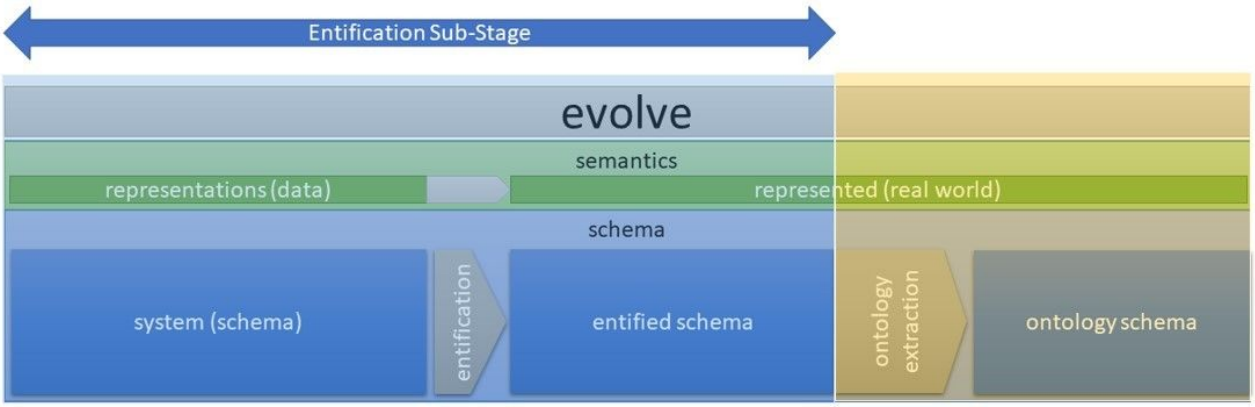


2.2.3 anatomy of a typical multi system bclearer process



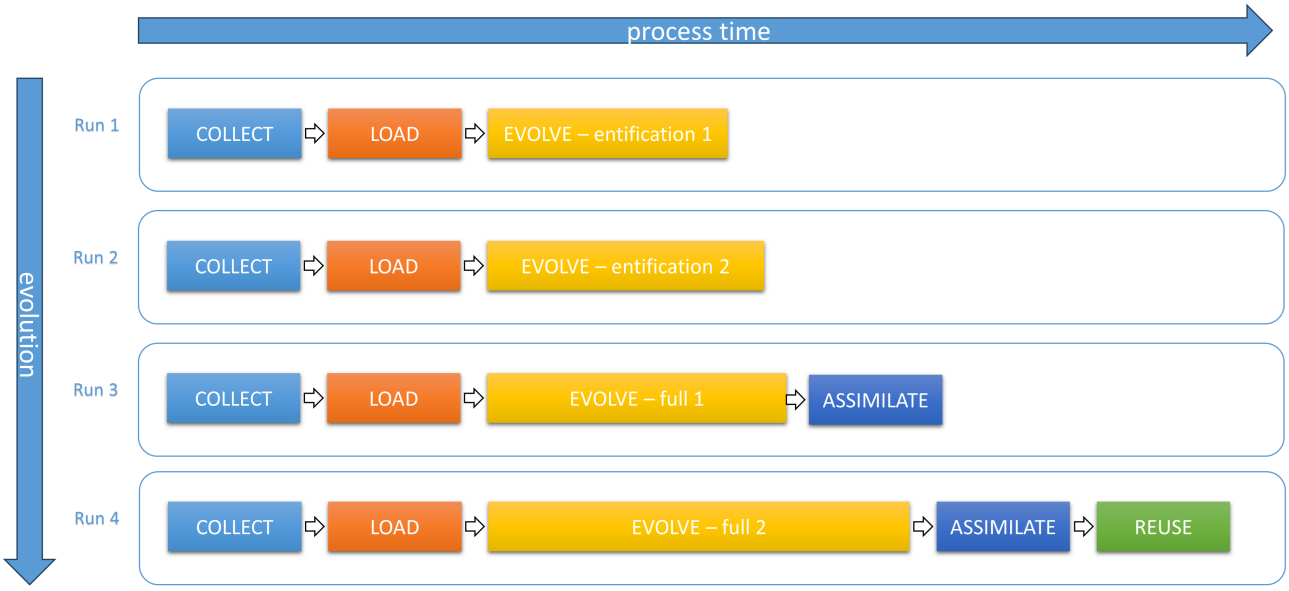
anatomy of a typical multi system bclearer process

2.2.4 anatomy of the bclearer process - entification substage

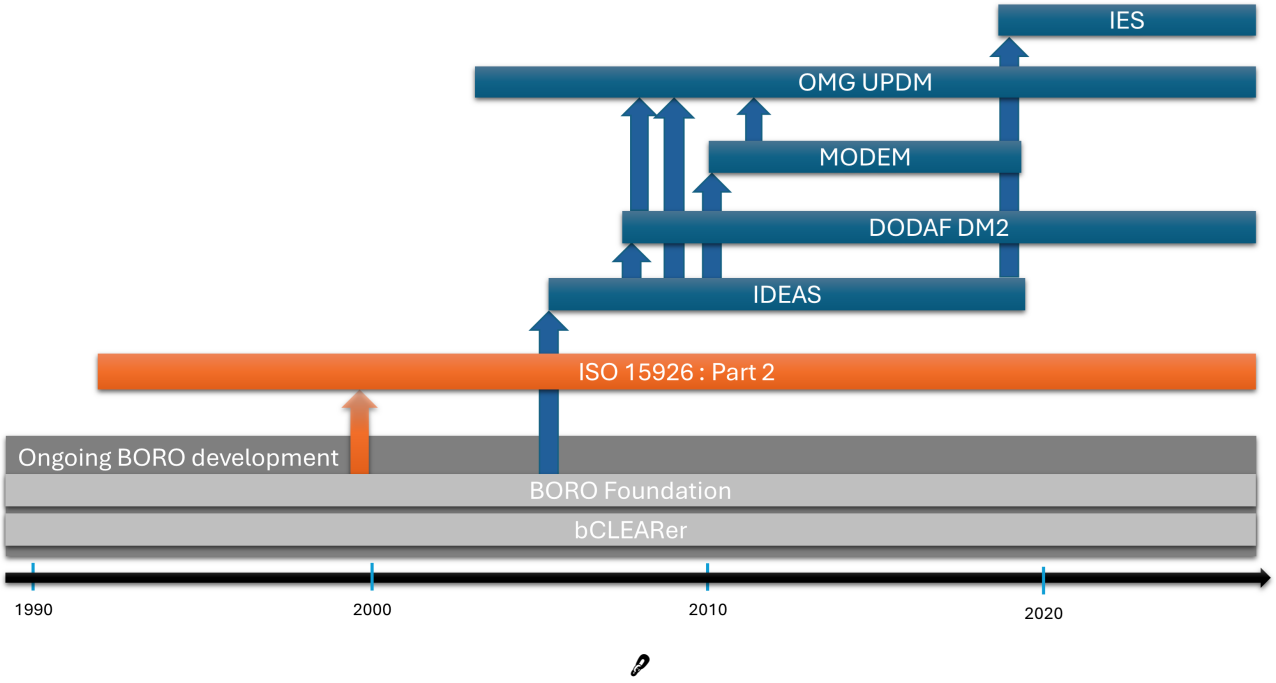


anatomy of the bclearer process - entification substage

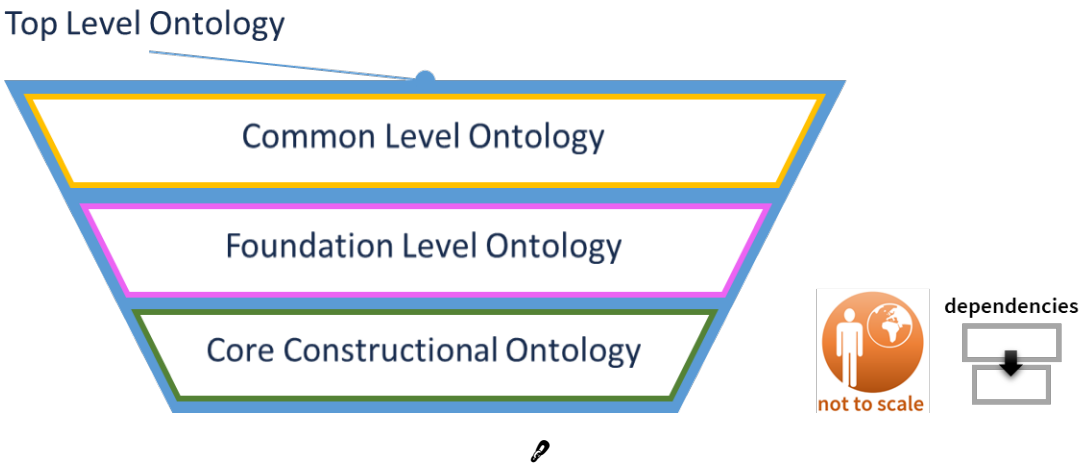
2.2.5 bclearer flow patterns - simple evolution



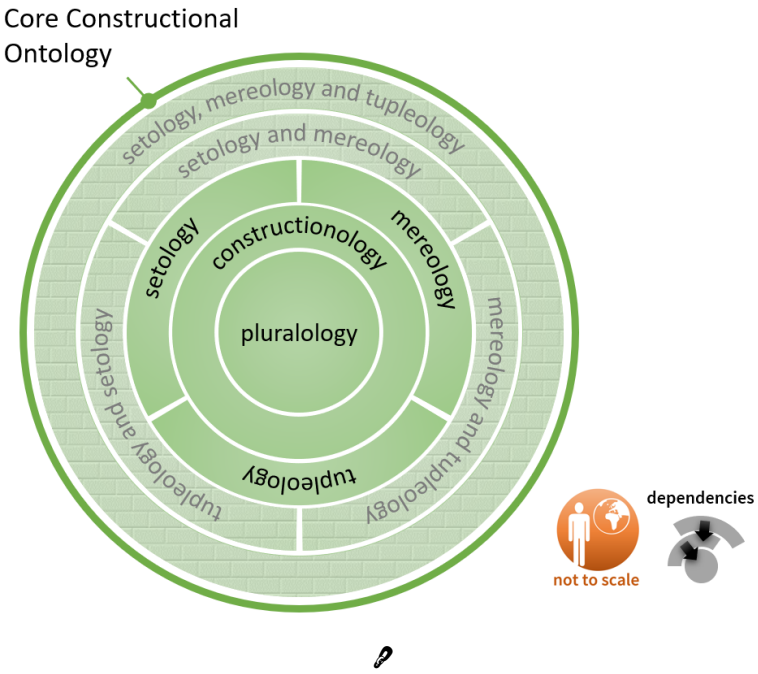
2.2.6 boro - bclearer standards - history



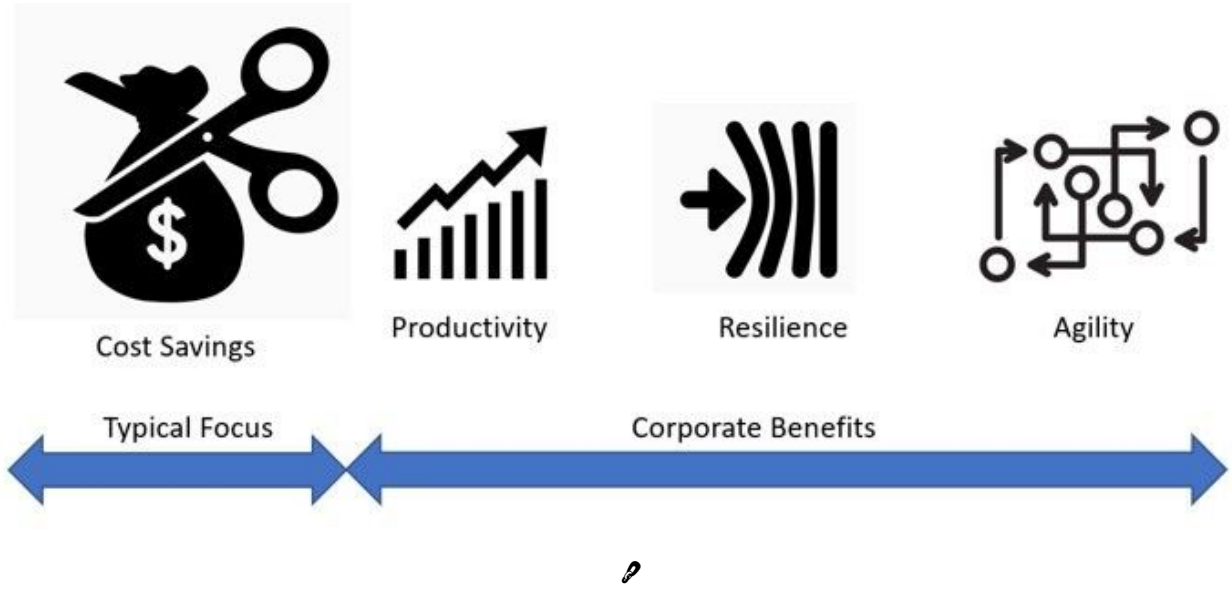
2.2.7 boro foundation - top level ontology



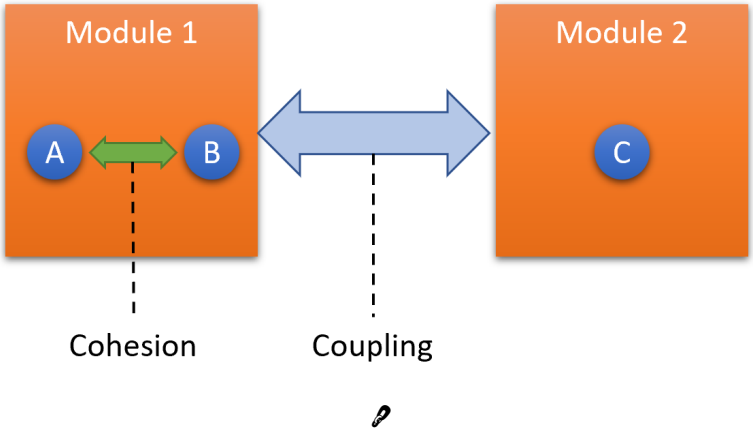
2.2.8 core constructional ontology



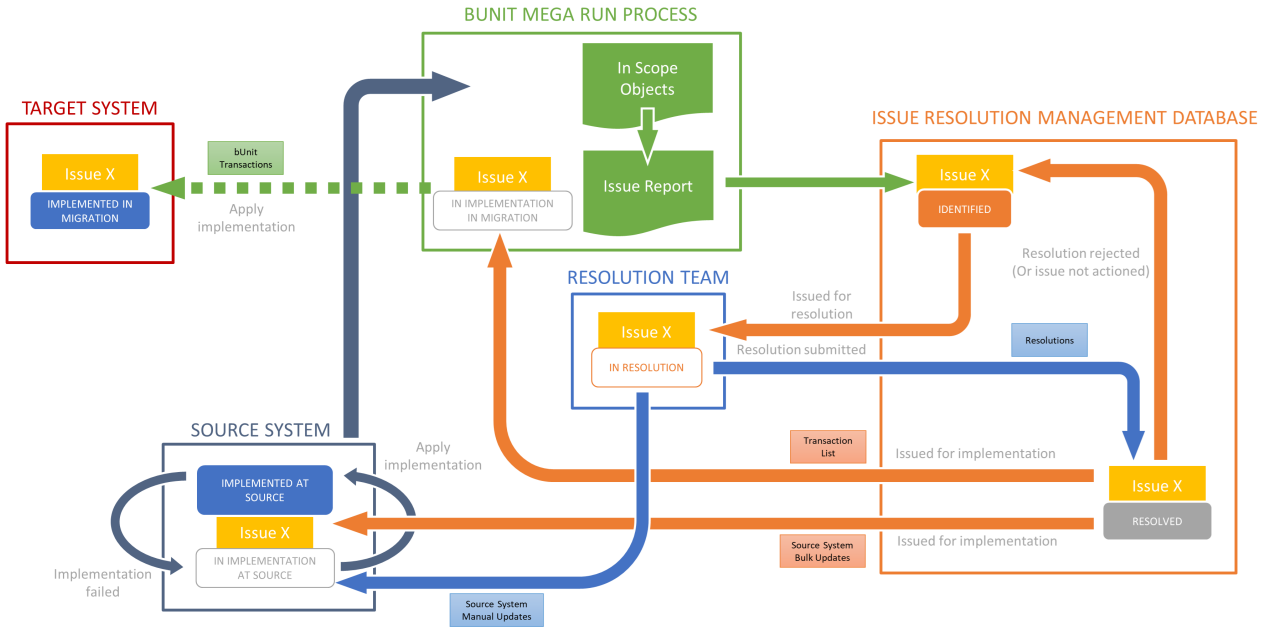
2.2.9 costs and benefits



2.2.10 coupling and cohesion

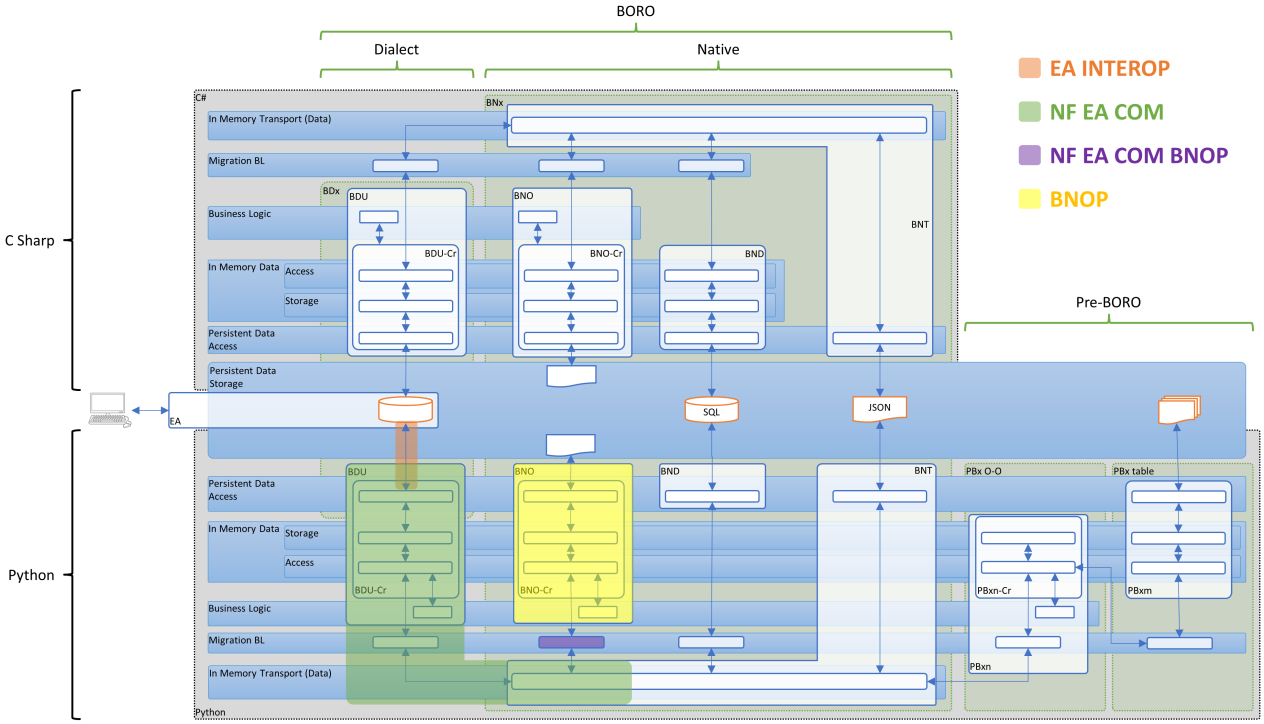


2.2.11 example - query resolution management process



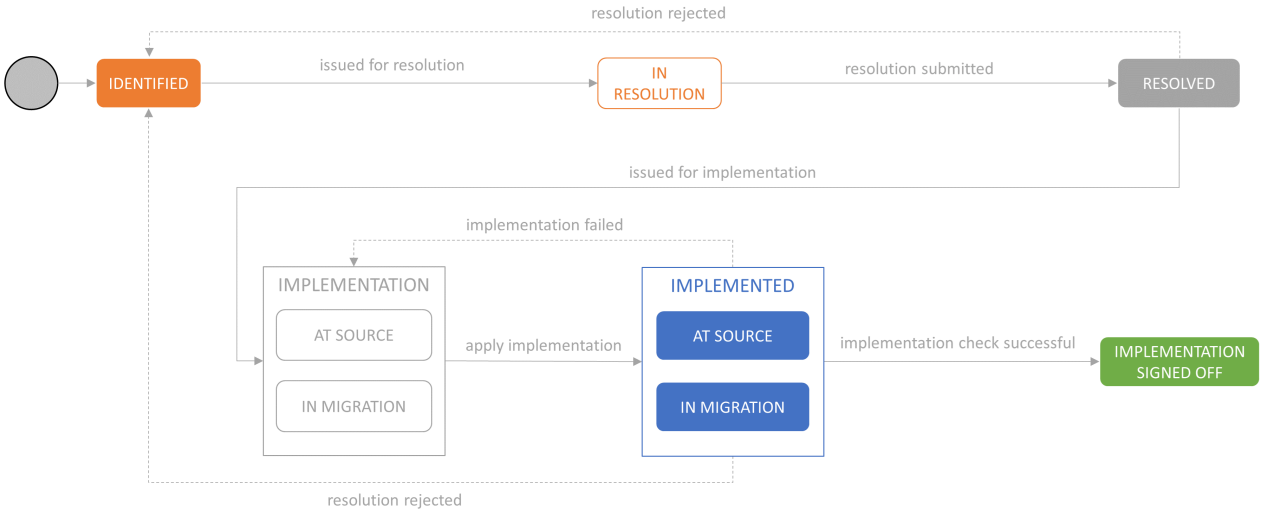
example - query resolution management process

2.2.12 historic nf architecture - with library footprints



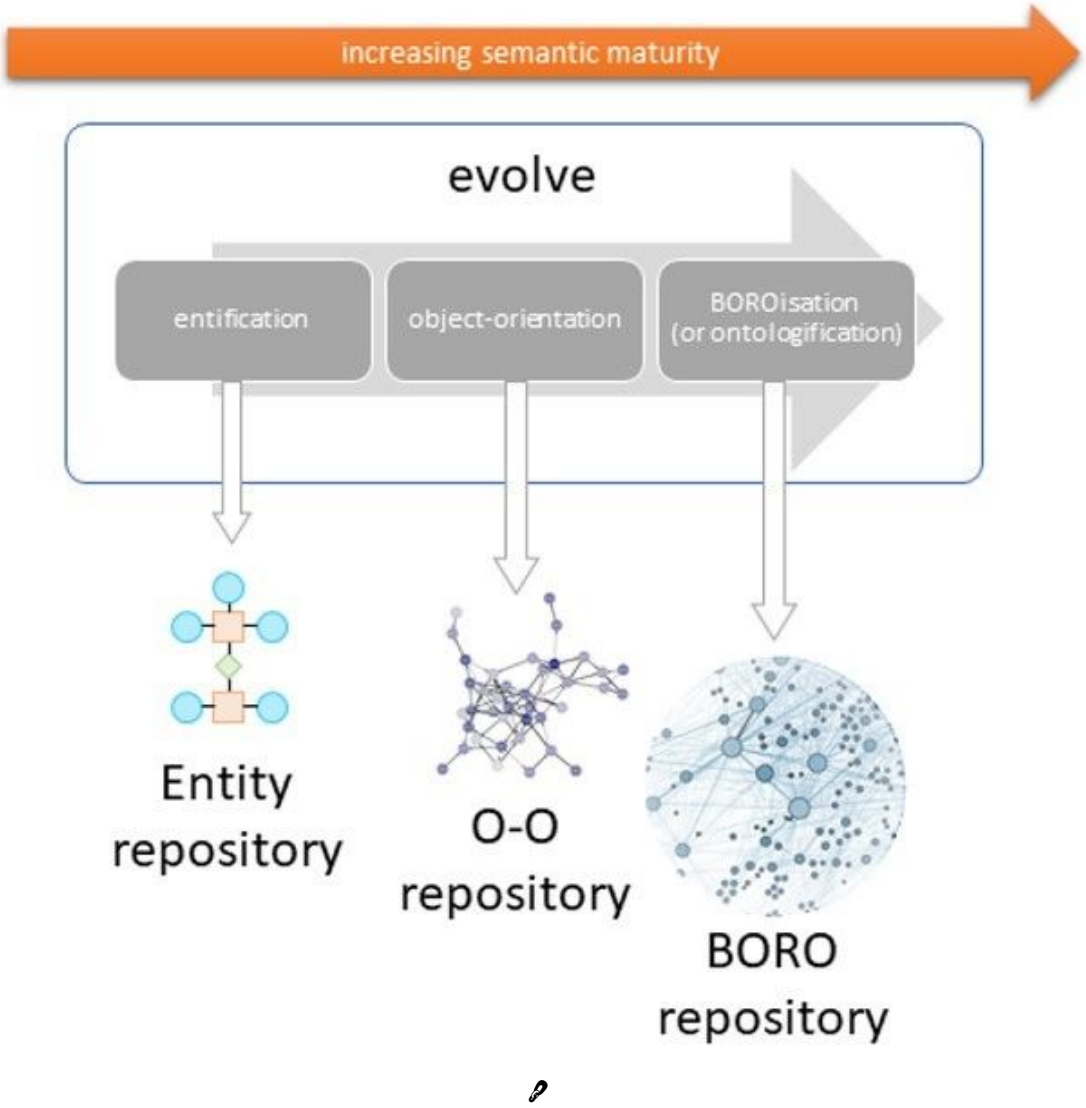
historic nf architecture - with library footprints

2.2.13 issue resolution workflow

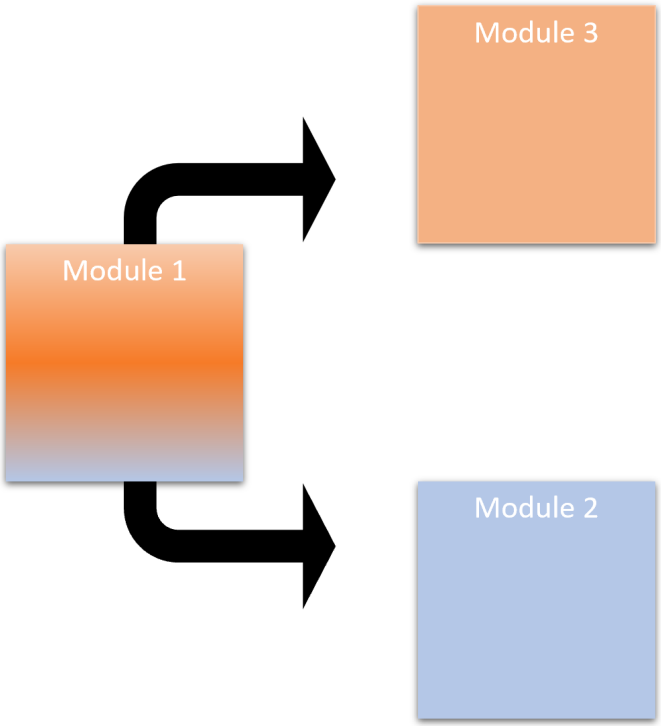


issue resolution workflow

2.2.14 natural truncation points in the bclearer process

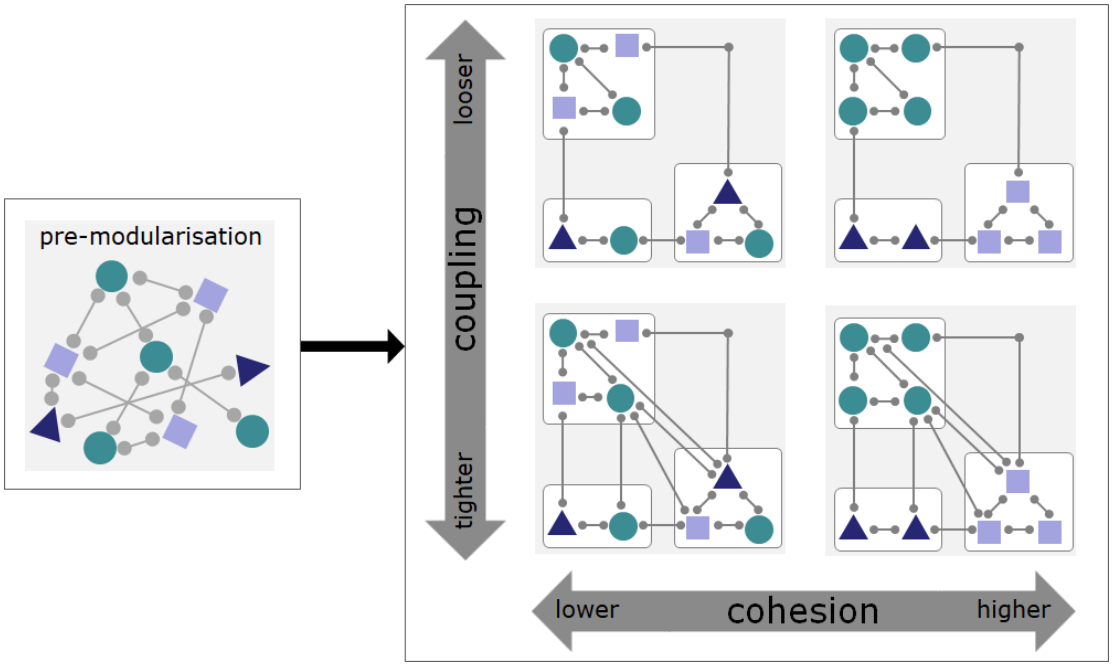


2.2.15 separating two concerns into two modules



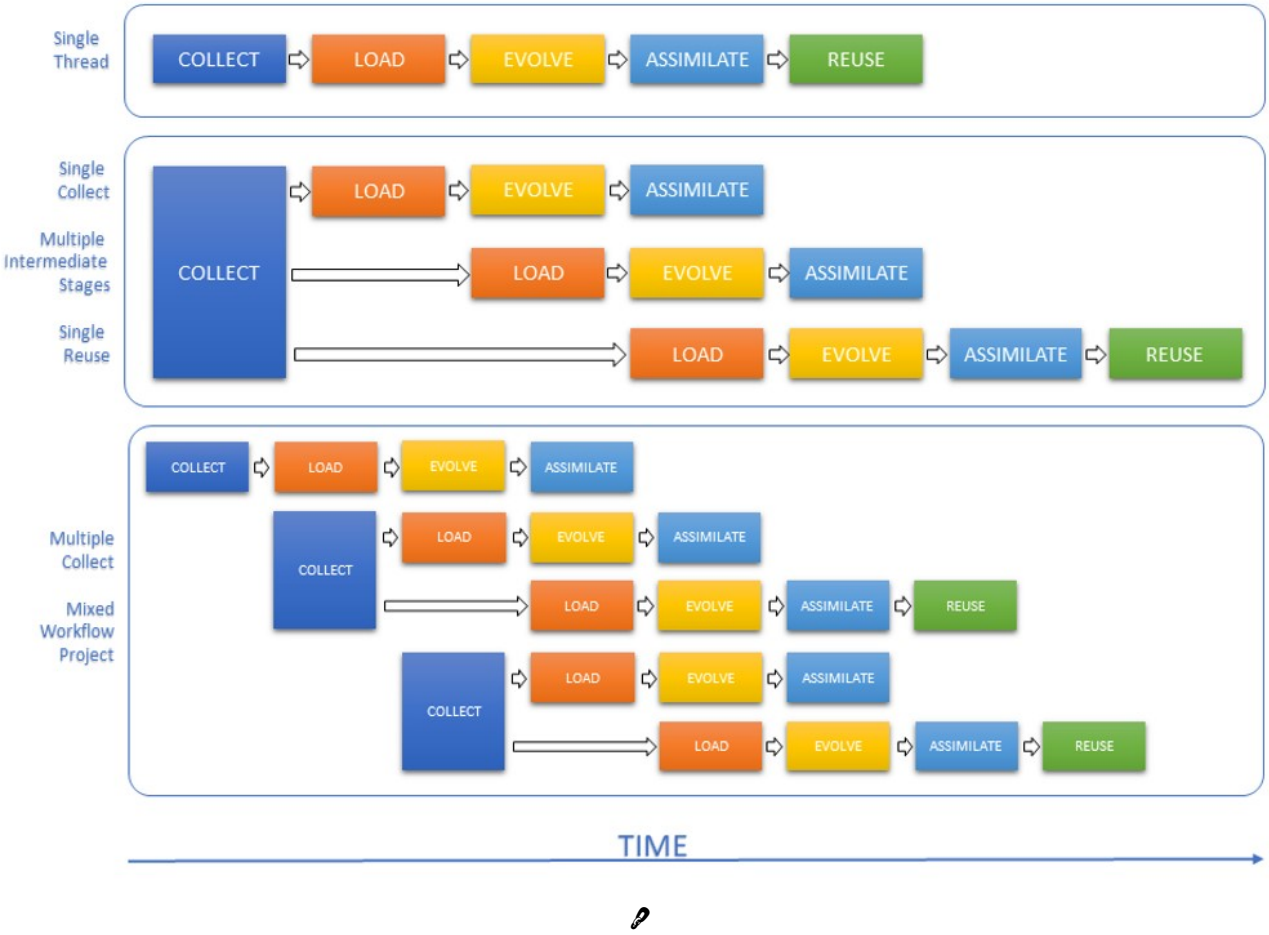
separating two concerns into two modules

2.2.16 shifting to loose coupling and tight cohesion

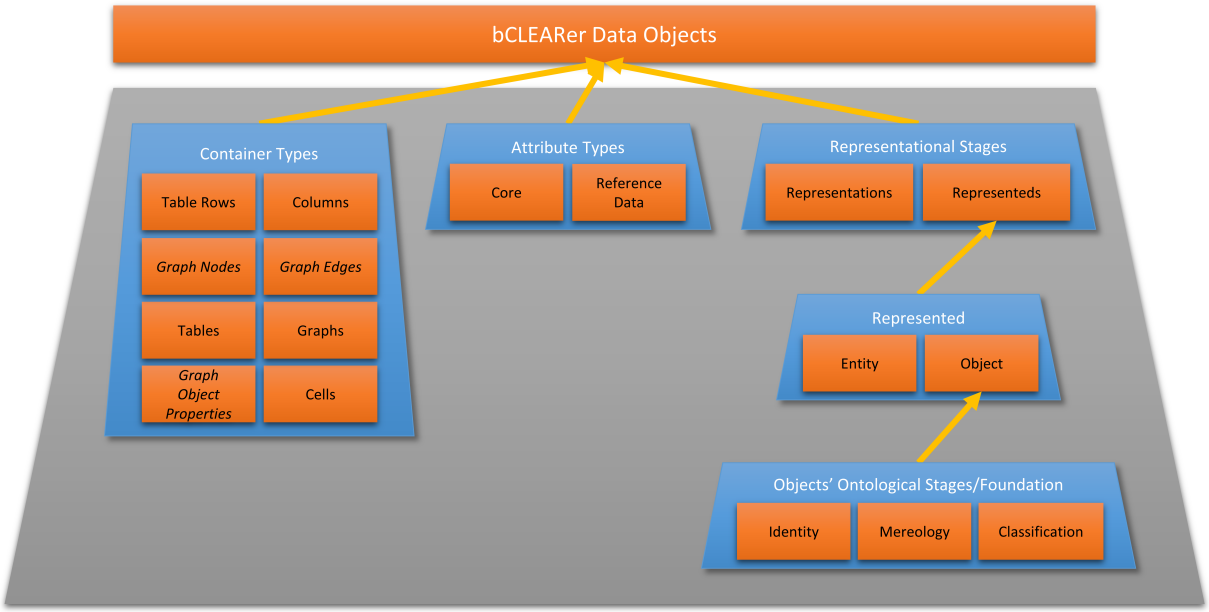


shifting to loose coupling and tight cohesion

2.2.17 typical bclearer flows

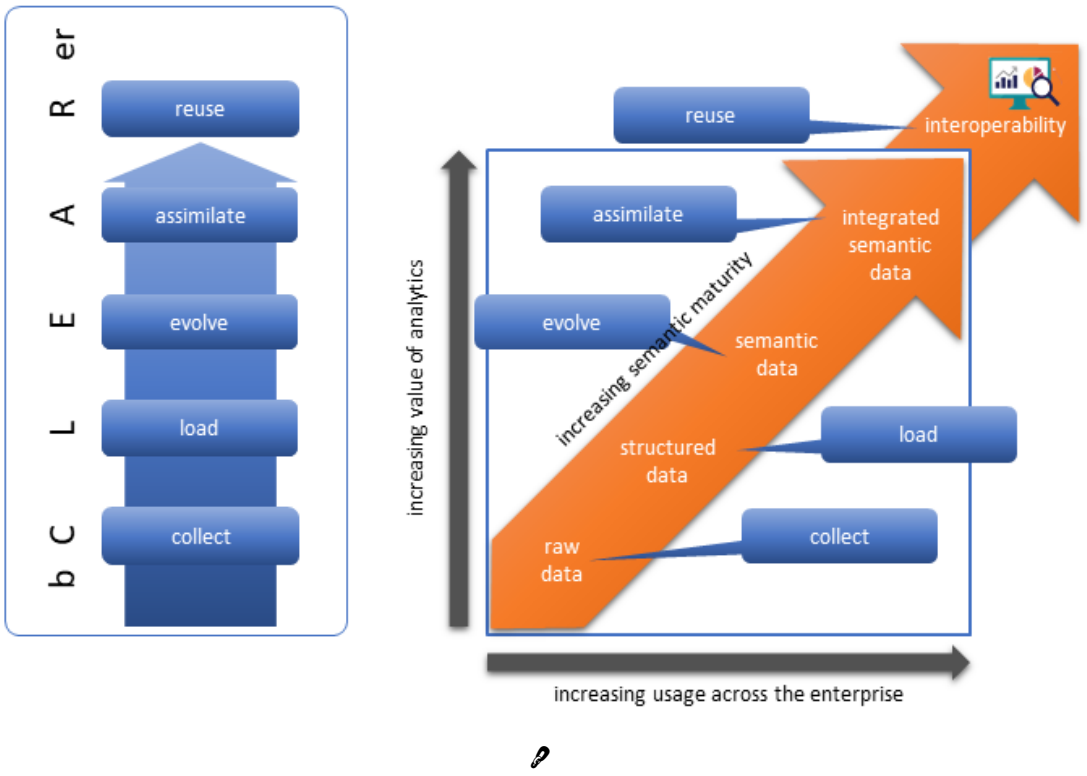


2.2.18 bclearer data object facets

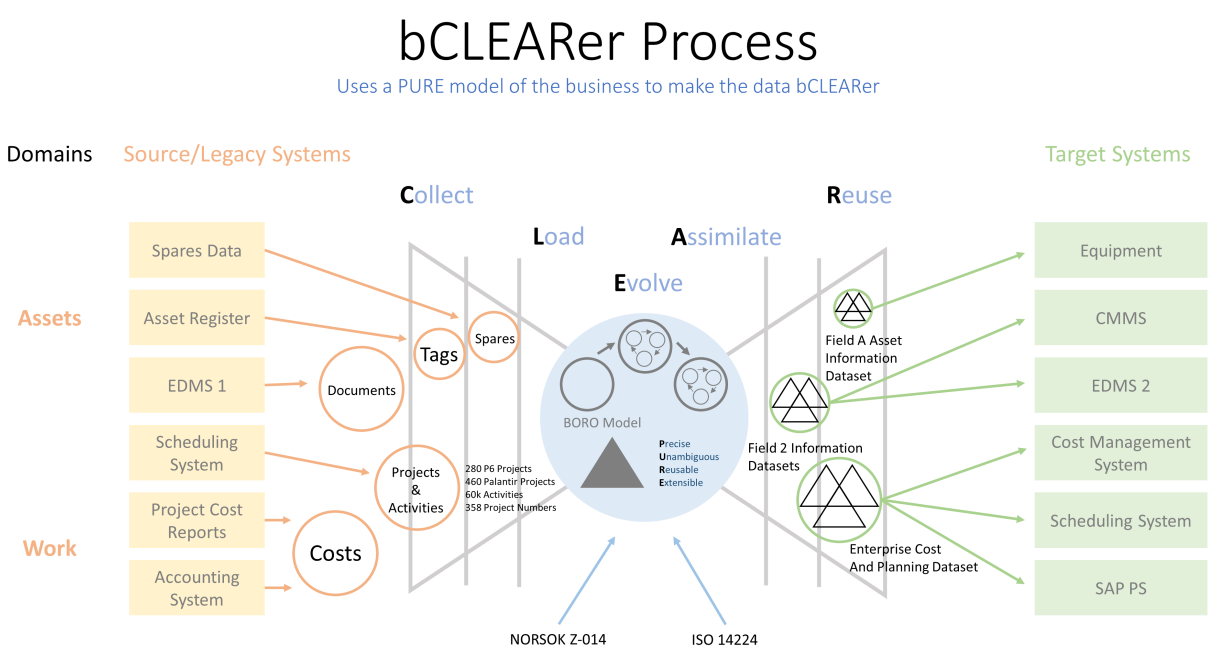


bclearer data object facets

2.2.19 mapping onto levels of semantic maturity

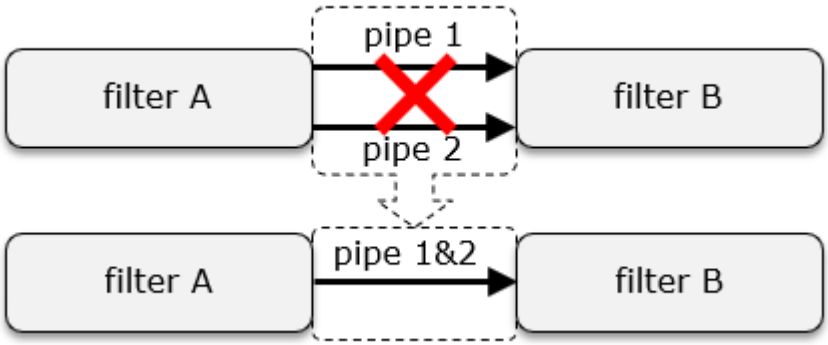


2.2.20 bclearer process - multi-system



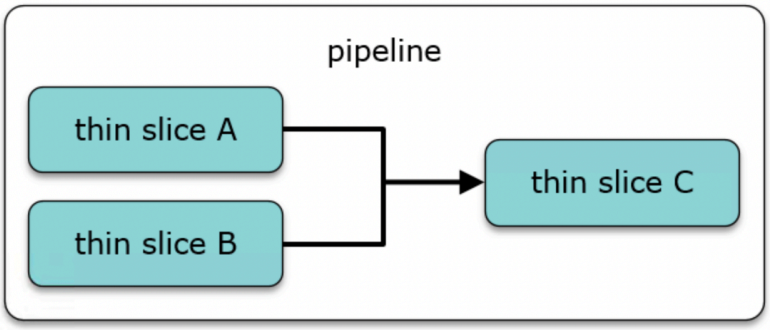
bclearer process - multi-system

2.2.21 multiple pipes between the same filters

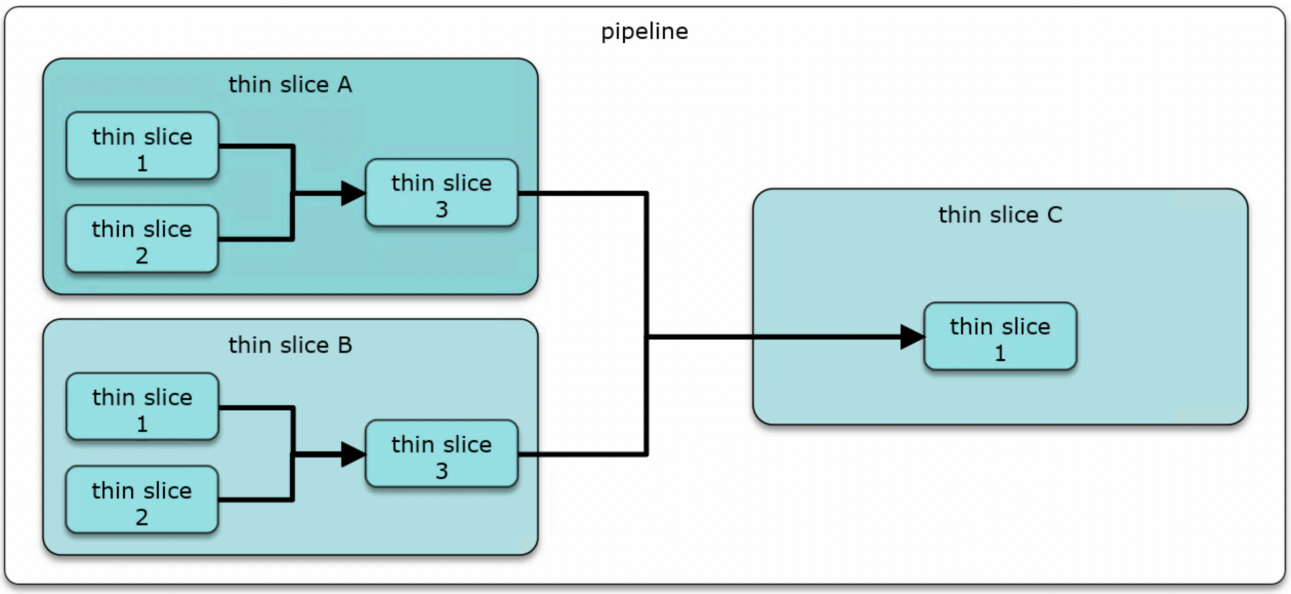


2.2.22 Resources - Graphics - Pipeline

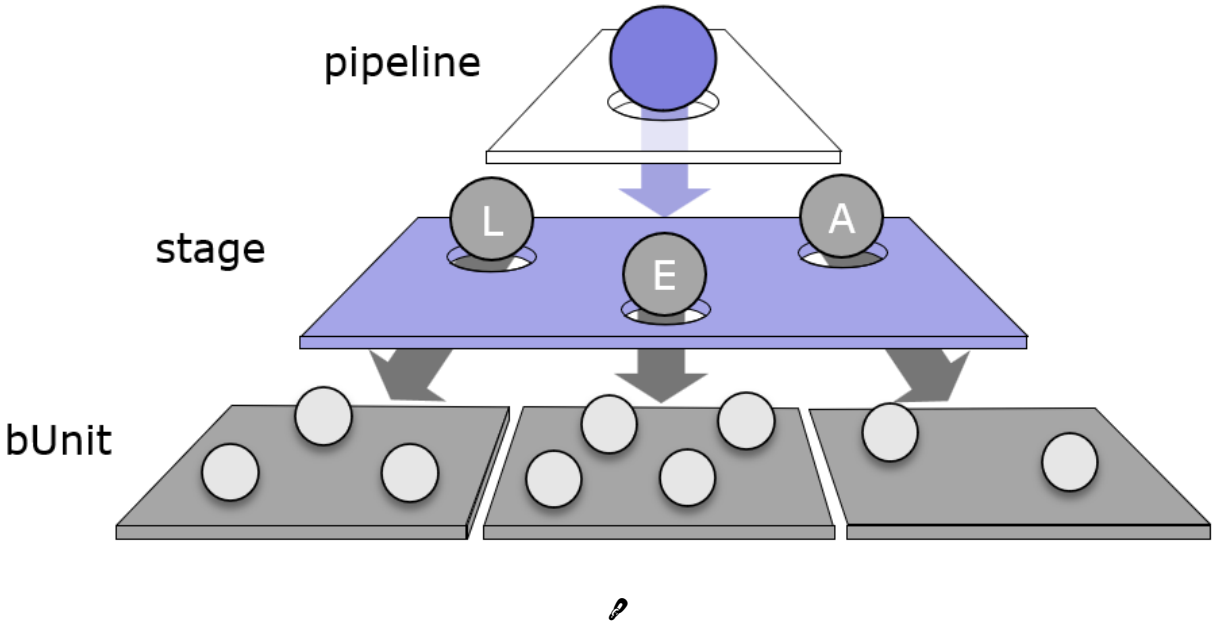
2.2.22.1 pro-forma example of a first stage decomposition



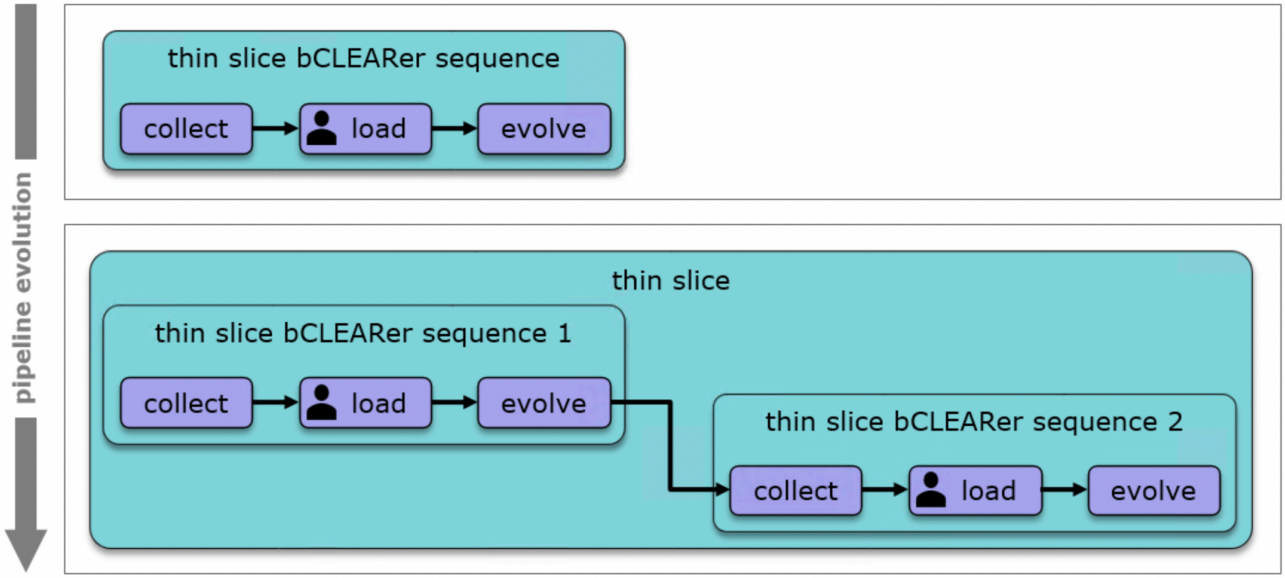
2.2.22.2 pro-forma nesting diagram - multiple layers



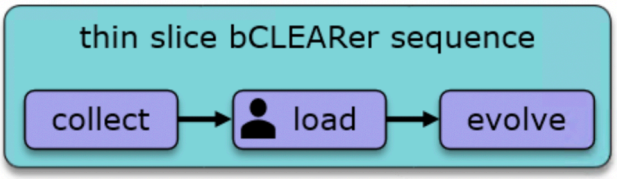
2.2.22.3 core breakdown levels



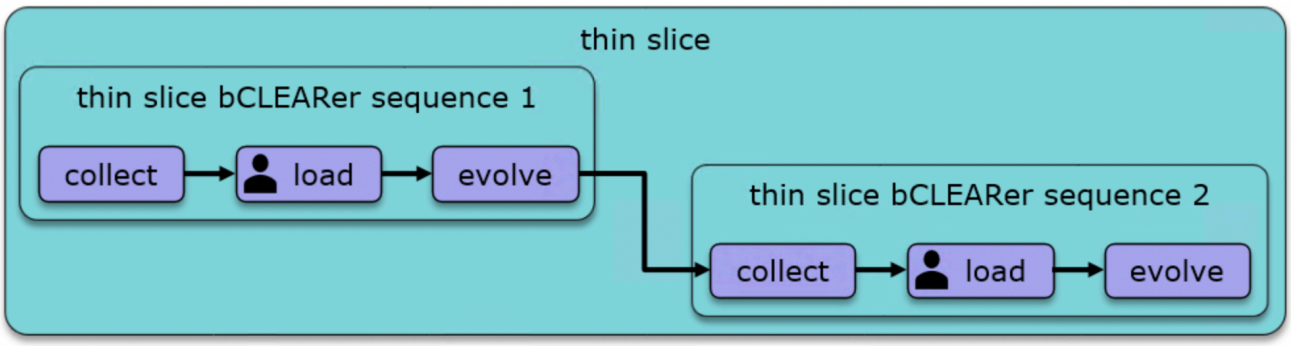
2.2.22.4 thin slice hierarchy evolution



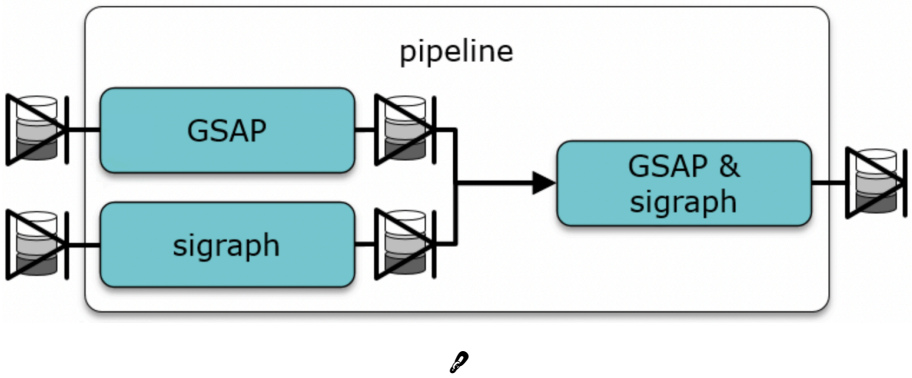
2.2.22.5 thin slice bCLEARer sequences - single



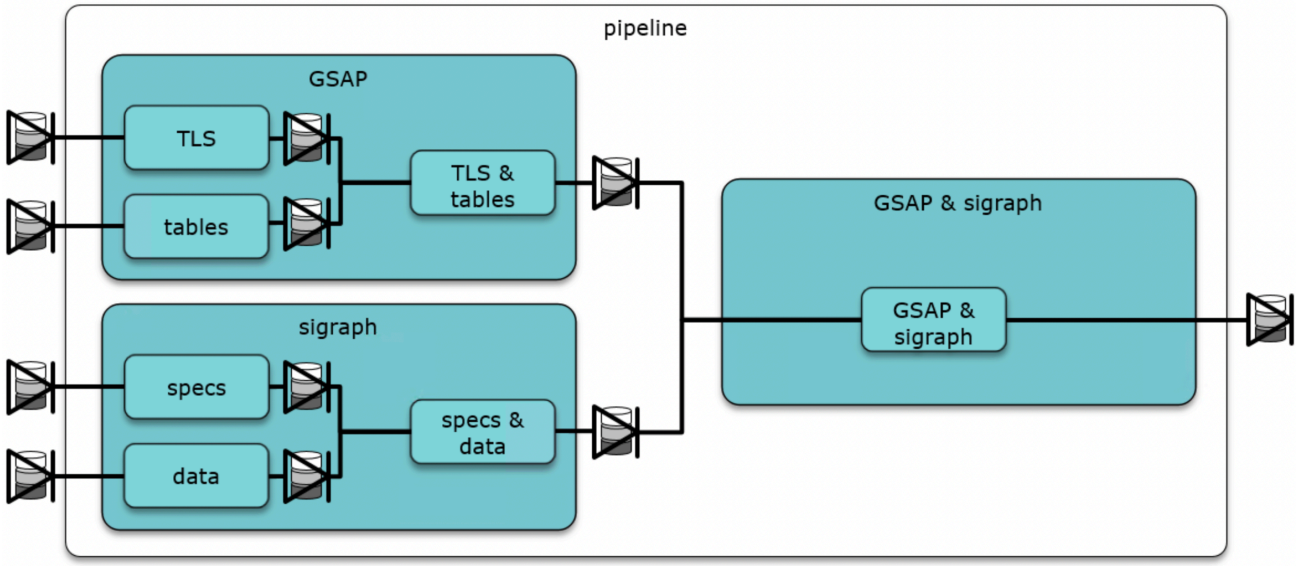
2.2.22.6 thin slice bCLEARer sequences - multiple



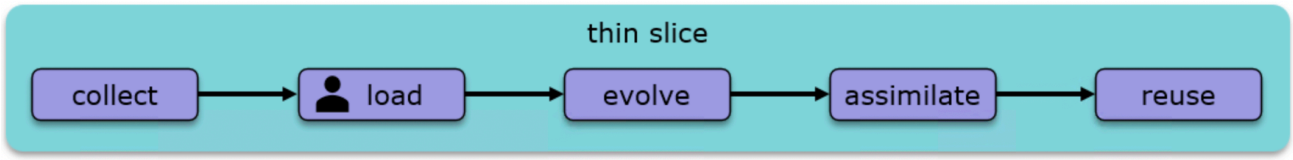
2.2.22.7 domain-based thin slicing



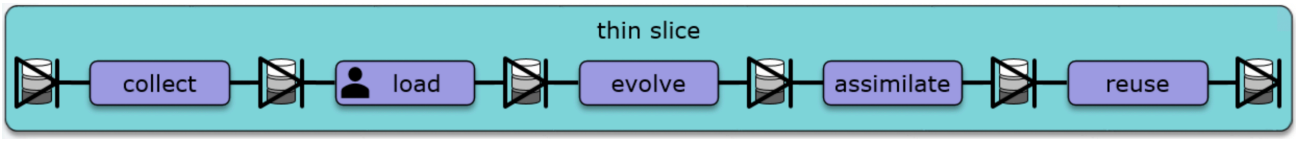
2.2.22.8 intra-domain thin slicing



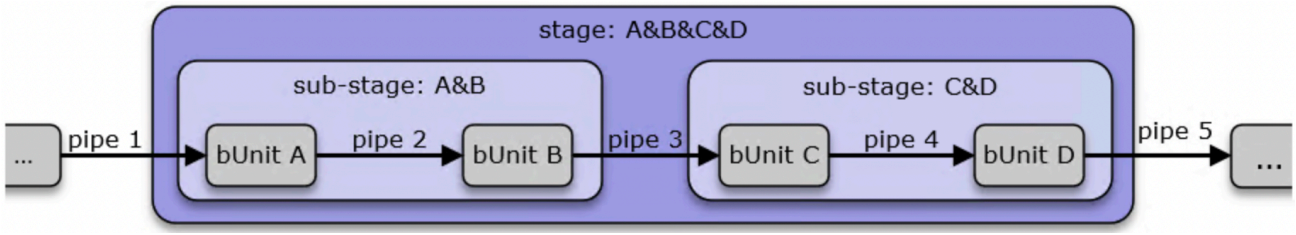
2.2.22.9 bCLEARer stages level



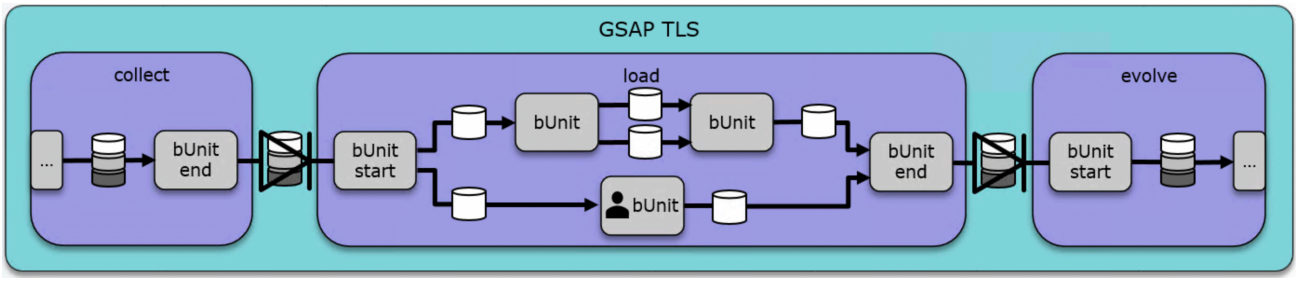
2.2.22.10 bCLEARer stages level - gated



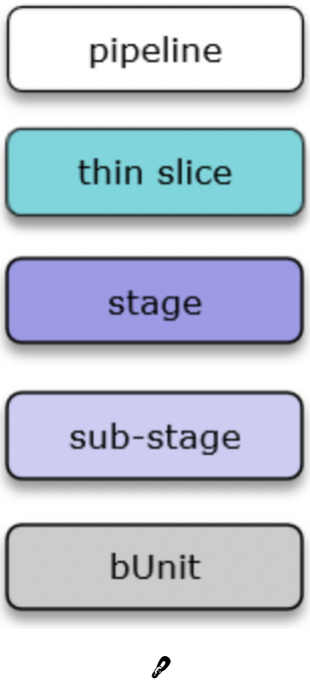
2.2.22.11 sub-stages level - grouping



2.2.22.12 stage gates



2.2.23 filter level type colouring legend



2.3 Resources - Other

2.3.1 (BORO Solutions Web Library)

The BORO Solutions web library archives presentations and papers produced for BORO projects in a searchable space.

A link to the library can be found here[78].

(see page 200)

78 https://www.borosolutions.net/boro-library

2.3.2 (Partridge - Academia profile)

A significant number of the presentations and papers produced for BORO projects were authored by Chris Partridge and can be found on https://www.academia.edu/. A good entry point is his academia profile page: https://westminster.academia.edu/ChrisPartridge.

2.3.3 bCLEARer Profiling Characteristics Lite

Good bCLEARer implementations have common characteristics. These could usefully be developed into a process for profiling them. We outline here the kinds of characteristics this profiling might look for. We use this to help us described the Code Examples (see page 80).

2.3.3.1 Pipeline characteristics

Characteristics	Details
pipeline 3 core level architecture	whether the 3 core level architecture is used (pipeline level, bCLEARer stage level, bUnit level). See Architectural nesting breakdown (see page 21)
pipeline bCLEARer stages	whether bCLEARer stages are used and if so which ones. See Stages level (see page 26)
pipeline bUnits	whether the lowest level base units of the process are bUnits that contain functions with a specific and atomic purpose arranged in pipelines. See bUnits level (see page 27)
pipeline gates	whether the pipeline has gates where a snapshot of the project may be reported. See Nested pipeline's data stage gates (see page 18)
pipeline topology	how lattice-like is the pipeline (topology); for example, are the bCLEARer stages single or multi-threaded?
pipeline evolution	whether the repo history shows pipeline evolution See Evolutionary time (see page 8)

2.3.3.2 Other characteristics

Characteristics	Details
bCLEARer pipeline workflow management	whether it is coded (uses standard code patterns) for rapid evolutionary development, or it uses a workflow management system (involving artifacts like configuration files and the code infrastructure needed to read and run the configuration) to enable configurability.

Characteristics	Details
bie object identity	whether the pipeline uses object identity. whether the pipeline uses the bie classes for object identity.
bCLEARer universes	whether the process uses any of the bCLEARer Universe classes. whether these classes are used for data storage and migration. whether the process uses a single or multiple universes.
bCLEARer query management	whether the pipeline has query management functionality. whether the queries and query types have object identity.
BORO bCLEARer common libraries	whether the pipeline uses the these libraries - and which libraries it uses. See Library Examples (see page 87)
BORO clean coding	whether the pipeline code follows the BORO Clean Coding Principles (see page 104).

2.3.4 Example - bCLEARer Process - bOSON 1.1 (bCLEARer Ordnance Survey Open Names) Surface Onomatology

The bOSON 1.1 (bCLEARer Ordnance Survey Open Names) Surface Onomatology[79] GitHub repository contains an example of the bCLEARer process implemented in Python that has been applied to the Ordnance Survey Open Names[80] dataset. At a more general level, this code provides an example of how to automatically implement (at scale) a surface onomatology pattern over structured data.

The Python code transparently documents the transformations that arise from the data cleaning and ontological analysis that takes place in the different stages of the bCLEARer process, and it outputs the result in a self-contained folder system divided into sub-folders for each of the bCLEARer stages and sub-stages.

As a standard bCLEARer inspection practice, output was provided for each of the sub-stages. This enabled the user to check the progress of the process.

This project applied the Architectural nesting breakdown (see page 21). This nesting architecture was developed using standard code patterns to facilitate rapid evolutionary development.

The pipeline experienced rapid evolution. However, only the latest snapshot was copied to the Open Source repo, so the evolution doesn't show in the repo's history.

The latest snapshot has a single bCLEARer sequence implementing the Stages level (see page 26) level pattern for Collect, Load and Evolve.

The Collect stage data (OS Open Names GML source data) can be downloaded online here[81].

The repo contains code for the Load (bCLEARer Load orchestrator[82]) and Evolve (bCLEARer Evolve orchestrator[83]) stages.

The bCLEARer sub-stages orchestrate a series of atomic single-purposed bUnits level (see page 27) (this Load substage code[84] shows a sequence of bUnit operations).

Object identity is maintained using UUIDS complying with the RFC 4122[85] standard (in this Load bUnit code example[86] we can see how a new object is given a UUID) which allows the tracking of objects throughout the process.

Object identity based bCLEARer Universes (see page 103) (NfEaComUniverses[87] class) are used to store and transport data. Snapshots of the project's universe are taken after each bUnit to maximise inspectability (as we can see in this Evolve substage code example[88]).

79 https://github.com/boro-alpha/bclearer_boson_1_1
80 https://www.ordnancesurvey.co.uk/products/os-open-map-local
81 https://osdatahub.os.uk/downloads/open/OpenNames
82 https://github.com/boro-alpha/bclearer_boson_1_1/blob/master/bclearer_boson_1_1_source/b_code/orchestrators/boson_1_load_stage_orchestrator.py
83 https://github.com/boro-alpha/bclearer_boson_1_1/blob/master/bclearer_boson_1_1_source/b_code/orchestrators/boson_1_evolve_stage_orchestrator.py
84 https://github.com/boro-alpha/bclearer_boson_1_1/blob/master/bclearer_boson_1_1_source/b_code/substages/visualization_substages/boson_1_visualization_substage_load_gml_data_runner.py
85 https://datatracker.ietf.org/doc/html/rfc4122
86 https://github.com/boro-alpha/bclearer_boson_1_1/blob/master/bclearer_boson_1_1_source/b_code/substages/operations/a_load/boson_1_gml_data_loader/os_open_names_domain/os_open_names_appenders/b1_os_open_names_dictionary_appender.py
87 https://github.com/boro-alpha/nf_ea_common_tools/blob/master/nf_ea_common_tools_source/b_code/services/general/nf_ea/com/nf_ea_com_universes.py
88 https://github.com/boro-alpha/bclearer_boson_1_1/blob/master/bclearer_boson_1_1_source/b_code/substages/visualization_substages/boson_1_visualization_substage_separate_names_and_instances_runner.py

There are object identity universe-based **gates** at the end of bCLEARer stages and sub-stages (as we can see in this Evolve substage code example[89]). This enabled common code to be used to visualise the gate snapshots of the universes and so inspect the process at each stage (common bCLEARer substage visualizer[90]).

This project did not require bCLEARer Query Architecture (see page 100) or reporting.

A significant proportion of the code is reused from these **BORO common libraries**: NF Common (see page 93), NF EA Common Tools (see page 89), EA Interop Service (see page 95), bCLEARer (see page 88), BNOP (see page 97), NF EA Com BNOP (see page 91), BORO Common (see page 94).

As is standard BORO practice, this project applied BORO Clean Coding Principles (see page 104).

This project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

 (see page 204)

89 https://github.com/boro-alpha/bclearer_boson_1_1/blob/master/bclearer_boson_1_1_source/b_code/substages/visualization_substages/boson_1_visualization_substage_separate_names_and_instances_runner.py

90 https://github.com/boro-alpha/bclearer/blob/master/bclearer_source/b_code/substages/visualizations/instrumentation_and_visualization_runner.py

2.3.5 Example - bCLEARer Process - bOSON 1.2 (bCLEARer Ordnance Survey Open Names) Deep Onomatology - Coordinates

The bOSON 1.2 (bCLEARer Ordnance Survey Open Names) Deep Onomatology[91]GitHub repository contains an example of the bCLEARer process implemented in Python that has been applied to the Ordnance Survey Open Names[92] dataset and the INSPIRE[93] metamodel. At a more general level, this code provides an example of how to automatically implement (at scale) a deep onomatology pattern over structured data.

The Python code transparently documents the transformations that arise from the data cleaning and ontological analysis that takes place in the different stages of the bCLEARer process, and it outputs the result in a self-contained folder system divided into sub-folders corresponding to each of the bCLEARer stages and sub-stages.

As a standard bCLEARer inspection practice, output was provided for each of the sub-stages. This enabled the user to check the progress of the process.

This project applied the Architectural nesting breakdown (see page 21). This nesting architecture was developed using standard code patterns to facilitate rapid evolutionary development.

The pipeline experienced rapid evolution. However, only the latest snapshot was copied to the Open Source repo, so the evolution doesn't show in the repo's history.

The latest snapshot has a single bCLEARer sequence implementing the Stages level (see page 26) level pattern for Collect, Load and Evolve.

The Collect stage data can be downloaded online; the OS Open Names GML from here[94], and the INSPIRE metamodel source data from here[95].

The repo contains code for the Load (bCLEARer Load orchestrator[96]) and Evolve (bCLEARer Evolve orchestrator[97]) stages.

The bCLEARer sub-stages orchestrate a series of atomic single-purposed bUnits level (see page 27) (this Evolve substage code[98] shows a sequence of bUnit operations).

Object identity is maintained using UUIDS complying with the RFC 4122[99] standard (example here[100]) which allows the tracking of objects throughout the process.

91 https://github.com/boro-alpha/bclearer_boson_1_2
92 https://www.ordnancesurvey.co.uk/products/os-open-map-local
93 https://inspire.ec.europa.eu/
94 https://osdatahub.os.uk/downloads/open/OpenNames
95 https://inspire.ec.europa.eu/portfolio/data-models
96 https://github.com/boro-alpha/bclearer_boson_1_2/blob/master/bclearer_boson_1_2_source/b_code/orchestrators/boson_1_2_load_stage_orchestrator.py
97 https://github.com/boro-alpha/bclearer_boson_1_2/blob/master/bclearer_boson_1_2_source/b_code/orchestrators/boson_1_2_evolve_stage_orchestrator.py
98 https://github.com/boro-alpha/bclearer_boson_1_2/blob/master/bclearer_boson_1_2_source/b_code/substages/visualization_substages/boson_1_2_visualization_substage_expose_implicit_point_structure_and_names_runner.py
99 https://datatracker.ietf.org/doc/html/rfc4122
100 https://github.com/boro-alpha/bclearer_boson_1_2/blob/master/bclearer_boson_1_2_source/b_code/substages/operations/b_evolve/coordinate_lines/common/nf_uuid_keyed_on_name_prefix_dictionary_getter.py

Object identity based bCLEARer Universes (see page 103) (NfEaComUniverses[101] class) are used to store and transport data. Snapshots of the project's universe are taken in specific points to maximise inspectability (as we can see in this EVOLVE substage code example[102]).

There are object identity universe-based **gates** at the end of bCLEARer stages and sub-stages (as we can see in this EVOLVE substage code example[103]). This enabled common code to be used to visualise the gate snapshots of the universes and so inspect the process at each stage (common bCLEARer substage visualizer[104]).

This project did not require bCLEARer Query Architecture (see page 100) or reporting.

A significant proportion of the code is reused from these **BORO common libraries**: NF Common (see page 93), NF EA Common Tools (see page 89), EA Interop Service (see page 95), bCLEARer (see page 88), BNOP (see page 97), NF EA Com BNOP (see page 91), BORO Common (see page 94).

As is standard BORO practice, this project applied BORO Clean Coding Principles (see page 104).

This project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

(see page 206)

101 https://github.com/boro-alpha/nf_ea_common_tools/blob/master/nf_ea_common_tools_source/b_code/services/general/nf_ea/com/nf_ea_com_universes.py
102 https://github.com/boro-alpha/bclearer_boson_1_2/blob/master/bclearer_boson_1_2_source/b_code/substages/visualization_substages/boson_1_2_visualization_substage_coordinate_lines_runner.py
103 https://github.com/boro-alpha/bclearer_boson_1_2/blob/master/bclearer_boson_1_2_source/b_code/substages/visualization_substages/boson_1_2_visualization_substage_generalise_runner.py
104 https://github.com/boro-alpha/bclearer/blob/master/bclearer_source/b_code/substages/visualizations/instrumentation_and_visualization_runner.py

2.3.6 Example - bCLEARer Process - UNICLASS bCLEARer

The UNICLASS bCLEARer Process[105] GitHub repository contains an example of the bCLEARer process (see bCLEARer - an introduction (see page 2)) applied implemented in Python that has been applied to the UNICLASS classification dataset[106].

The Python code transparently documents the transformations that arise from the data cleaning and ontological analysis that takes place in the different stages of the process, and it outputs the result in a self-contained folder system divided into sub-folders corresponding to each of the implemented bCLEARer stages and sub-stages.

As a standard bCLEARer inspection practice, output is provided for each of the sub-stages. This enables the user to check the progress of the process.

For each of the sub-stages, the user can check the progress of the analysis in three kinds of output files:

- domain tables: collection of data tables showing a standard human-readable structure (as CSV and Access databases)
- nf ea com tables: collection of data tables which are the result of processing the domain tables to give them a more UML-friendly structure (influenced by the Enterprise Architect Data Model) - (as CSV and Access databases)
- Enterprise Architect[107] model files - useful for visualizing

This project applied the Architectural nesting breakdown (see page 21). This nesting architecture was developed using standard code patterns to facilitate rapid evolutionary development.

The pipeline experienced rapid evolution. However, only the latest snapshot was copied to the Open Source repo, so the evolution doesn't show in the repo's history.

The latest snapshot has a single bCLEARer sequence implementing the Stages level (see page 26) level pattern for Collect, Load and Evolve.

The Collect stage data (the UNICLASS classification dataset) can be downloaded online here[108]. The results of early Load are stored in the repository's Load resources folder[109].

The repo contains code for the Load (bCLEARer Load orchestrator[110]) and Evolve (bCLEARer Evolve orchestrator[111]) stages.

The bCLEARer sub-stages orchestrate a series of atomic single-purposed bUnits level (see page 27) (this Evolve substage code[112] shows a sequence of bUnit operations).

105 https://github.com/boro-alpha/uniclass_to_nf_ea_com
106 https://www.thenbs.com/our-tools/uniclass-2015
107 https://sparxsystems.com/products/ea/
108 https://www.thenbs.com/our-tools/uniclass-2015
109 https://github.com/boro-alpha/uniclass_to_nf_ea_com/tree/master/uniclass_to_nf_ea_com_source/c_resources/1_load_inputs
110 https://github.com/boro-alpha/uniclass_to_nf_ea_com/blob/master/uniclass_to_nf_ea_com_source/b_code/orchestrators/stages/load/load_stage_4_orchestrator.py
111 https://github.com/boro-alpha/uniclass_to_nf_ea_com/blob/master/uniclass_to_nf_ea_com_source/b_code/orchestrators/uniclass_bclearer_orchestrator.py
112 https://github.com/boro-alpha/uniclass_to_nf_ea_com/blob/master/uniclass_to_nf_ea_com_source/b_code/migrators/uniclass_raw_to_domain/evolve/evolve_stage_2/domain_tables_data_processor/evolve_stage_2_domain_tables_getter.py

Object identity is maintained using UUIDS complying with the RFC 4122[113] standard (we can see the creation of a UUID for a new object in this Evolve substage code example[114]) which allows the tracking of objects throughout the process.

Object identity based bCLEARer Universes (see page 103) (NfEaComUniverses[115] class) are used to store and transport data. Snapshots of the project's universe are taken in specific points to maximise inspectability (example here[116]).

There are object identity universe-based **gates** at the end of bCLEARer stages and sub-stages (as we can see in this EVOLVE substage code example[117]). This enabled common code to be used to visualise the gate snapshots of the universes and so inspect the process at each stage (common bCLEARer substage visualizer[118]).

This project did not require bCLEARer Query Architecture (see page 100) or reporting.

A significant proportion of the code is reused from these **BORO common libraries**: NF Common (see page 93), NF EA Common Tools (see page 89), EA Interop Service (see page 95).

As is standard BORO practice, this project applied BORO Clean Coding Principles (see page 104).

This project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

(see page 208)

113 https://datatracker.ietf.org/doc/html/rfc4122
114 https://github.com/boro-alpha/uniclass_to_nf_ea_com/blob/753e97467ce53c25bc86341b915489c2eeeb3f49/uniclass_to_nf_ea_com_source/b_code/migrators/uniclass_raw_to_domain/evolve/evolve_stage_5/domain_tables_data_processor/evolve_stage_5_domain_tables_getter.py
115 https://github.com/boro-alpha/nf_ea_common_tools/blob/master/nf_ea_common_tools_source/b_code/services/general/nf_ea/com/nf_ea_com_universes.py
116 https://github.com/boro-alpha/uniclass_to_nf_ea_com/blob/master/uniclass_to_nf_ea_com_source/b_code/migrators/uniclass_nf_ea_com_exporters/export_nf_ea_com_orchestrator.py
117 https://github.com/boro-alpha/uniclass_to_nf_ea_com/blob/master/uniclass_to_nf_ea_com_source/b_code/orchestrators/stages/evolve/evolve_stage_4_orchestrator.py
118 https://github.com/boro-alpha/bclearer/blob/master/bclearer_source/b_code/substages/visualizations/instrumentation_and_visualization_runner.py

2.3.7 Example - BORO Code Library - bCLEARer

> This is a historic snapshot of a working library that is under continual development.

This bCLEARer[119] GitHub repository is a Python reference library containing utility functions that are being used by other GitHub projects within the bCLEARer process developed by BORO Solutions[120].

Examples of the common code are:

- operations[121] - used in the execution of bUnits level (see page 27)
- visualizations[122] - used for gate inspection.

The code was developed using BORO Clean Coding Principles (see page 104).

> This GitHub project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

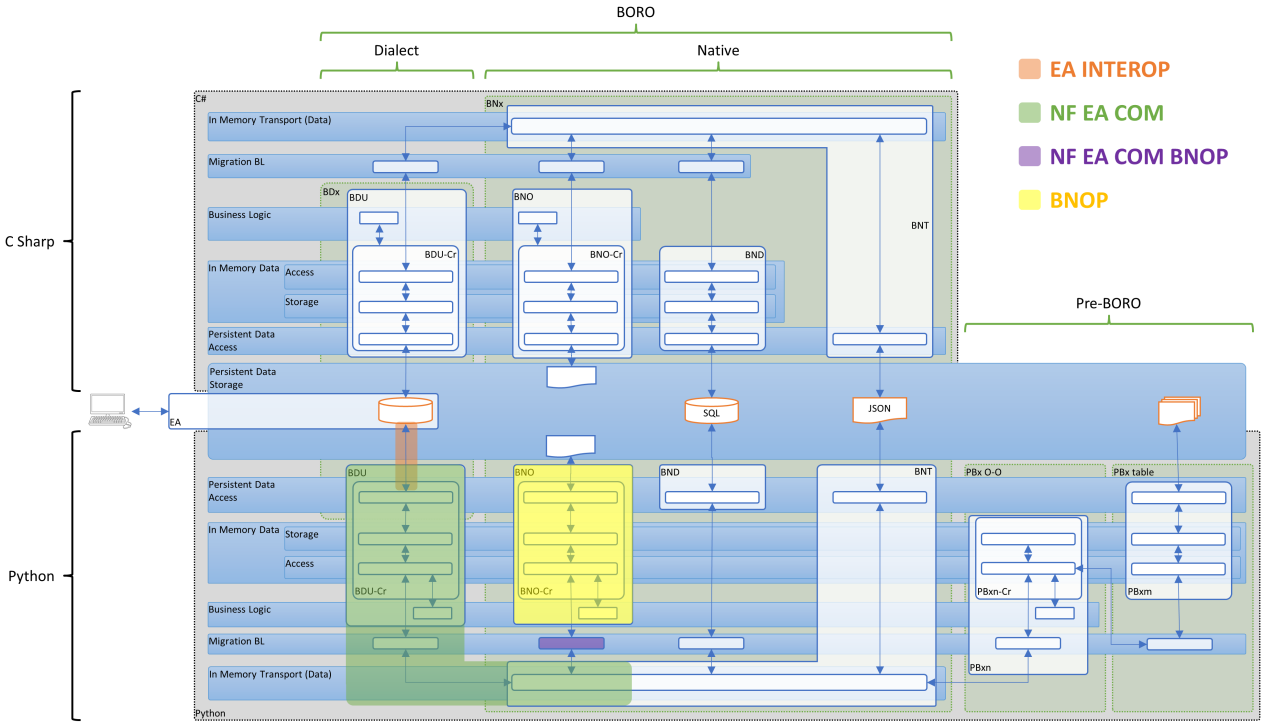
(see page 210)

119 https://github.com/boro-alpha/bclearer
120 https://borosolutions.net/
121 https://github.com/boro-alpha/bclearer/tree/master/bclearer_source/b_code/substages/operations
122 https://github.com/boro-alpha/bclearer/tree/master/bclearer_source/b_code/substages/visualizations

2.3.8 Example - BORO Code Library - BNOP

> This is a historic snapshot of a working library that is under continual development.

This BNOP[123] GitHub repository is a Python reference library containing the code for the BNOP Domain (BORO Native Objects (Python)) - this is an implementation of the BORO Top Ontology in Python. This has been implemented in a range of languages - as an historic NF architecture diagram below shows (the footprint of this BNOP library is highlighted - as are the other libraries).



historic nf architecture - with library footprints

The code in this library is being used by other GitHub projects within the BORO domain.

The code was developed using BORO Clean Coding Principles (see page 104).

> This GitHub project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

(see page 211)

123 https://github.com/boro-alpha/bnop

2.3.9 Example - BORO Code Library - BORO Common

> This is a historic snapshot of a working library that is under continual development.

This BORO Common[124] GitHub repository is a Python reference library containing enumerations that are being used by other GitHub projects within the BORO domain.

The code was developed using BORO Clean Coding Principles (see page 104).

> This GitHub project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

(see page 212)

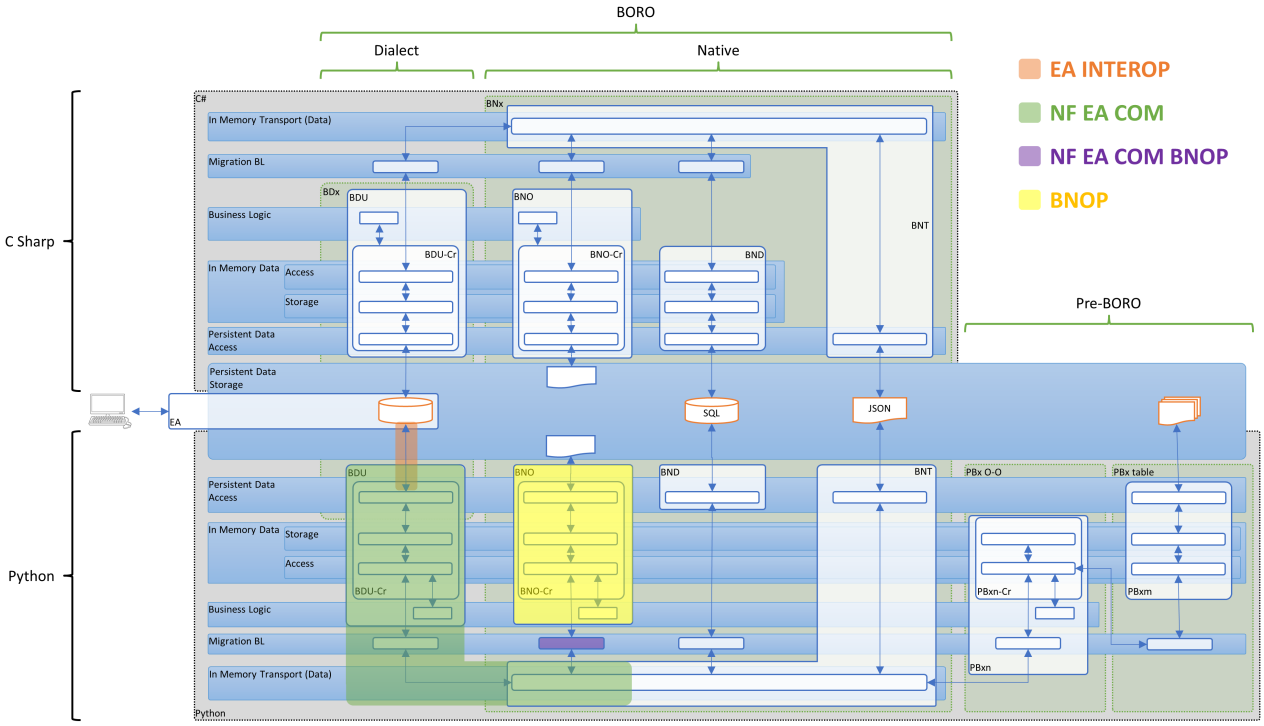
124 https://github.com/boro-alpha/boro_common

2.3.10 Example - BORO Code Library - EA Interop Service

> This is a historic snapshot of a working library that is under continual development.

This EA Interop Service[125] GitHub repository is a Python reference library that exposes parts of the Sparx EA object model[126] interface developed by BORO Solutions[127].

The historic NF architecture diagram below highlights the footprint of this EA Interop Service library along with the other libraries.



historic nf architecture - with library footprints

The code in this library is being used by other GitHub projects within the BORO domain.

The code was developed using BORO Clean Coding Principles (see page 104).

> This GitHub project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

125 https://github.com/boro-alpha/ea_interop_service
126 https://sparxsystems.com/enterprise_architect_user_guide/14.0/automation/theautomationinterface.html
127 https://borosolutions.net/

(see page 213)

2.3.11 Example - BORO Code Library - NF Common

> This is a historic snapshot of a working library that is under continual development.

This (New Foundations[128]) NF Common[129] GitHub repository is a Python reference library containing utility functions that are being used by other GitHub projects within the BORO domain.

The code was developed using BORO Clean Coding Principles (see page 104).

> This GitHub project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

(see page 215)

128 https://en.wikipedia.org/wiki/New_Foundations
129 https://github.com/boro-alpha/nf_common

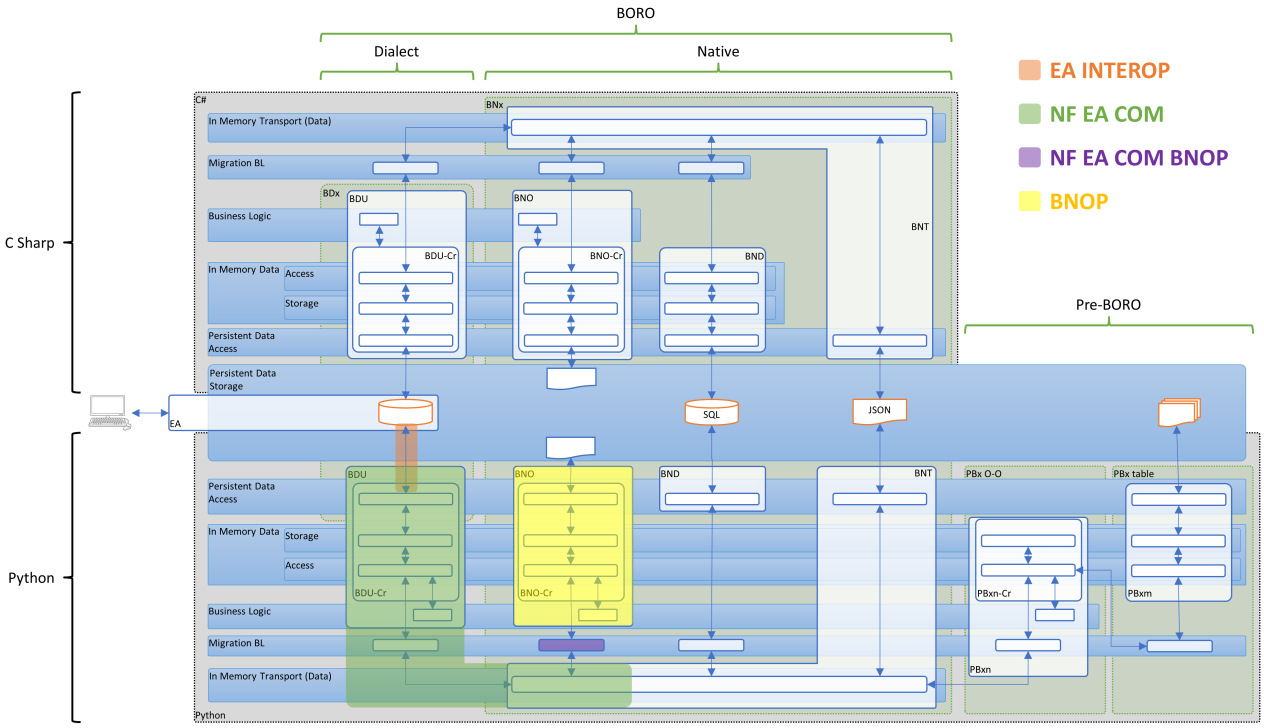
2.3.12 Example - BORO Code Library - NF EA COM BNOP

> This is a historic snapshot of a working library that is under continual development.

This NF EA COM BNOP[130] GitHub repository is a Python reference library containing utility functions that migrate between the NF EA COM domain and the BNOP[131] domain.

- The NF EA COM domain is an inter-domain data layer providing objects and functions to store the data in a format that aligns closely with EA (Enterprise Architect[132]). It can then be easily imported into and visualised.
- BNOP is the boro native objects (in Python)

The historic NF architecture diagram below highlights the footprint of this NF EA COM BNOP library along with the other libraries.



historic nf architecture - with library footprints

The code in this library is being used by other GitHub projects within the BORO domain.

The code was developed using BORO Clean Coding Principles (see page 104).

130 https://github.com/boro-alpha/nf_ea_com_bnop
131 https://github.com/boro-alpha/bnop
132 https://sparxsystems.com/products/ea/

This GitHub project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

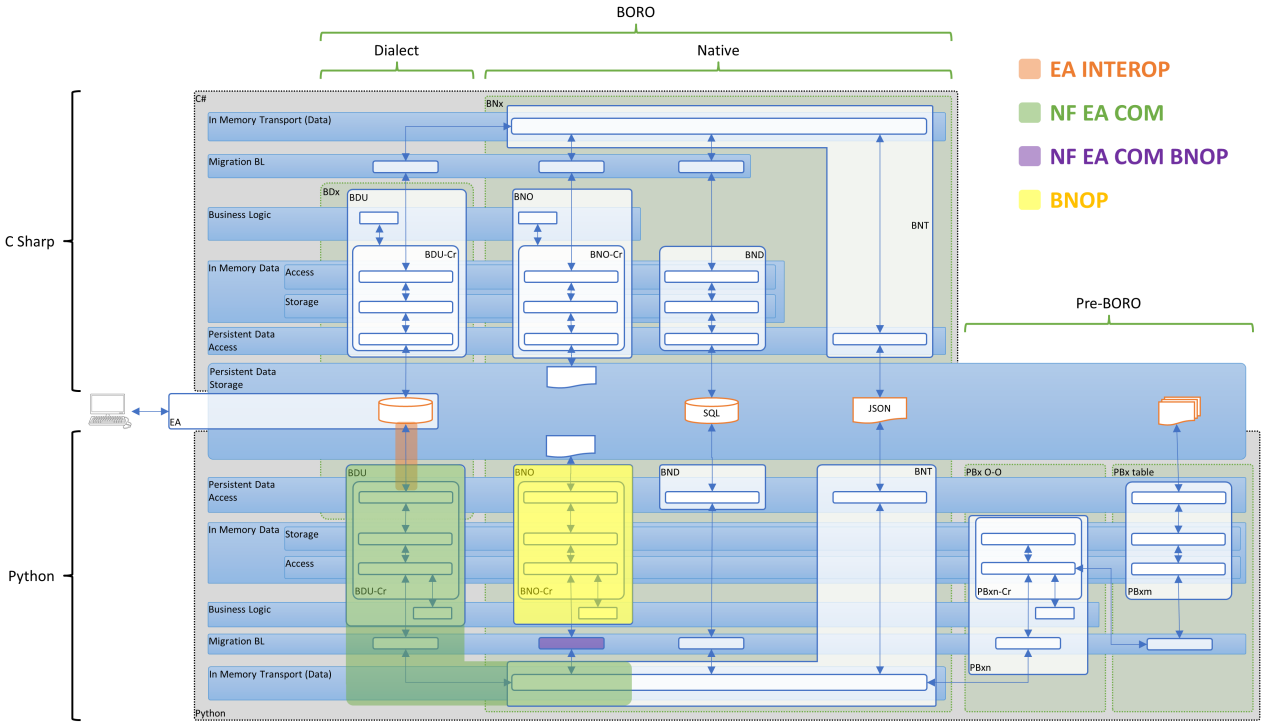
(see page 216)

2.3.13 Example - BORO Code Library - NF EA Common Tools

> This is a historic snapshot of a working library that is under continual development.

This NF EA Common Tools[133] GitHub repository is a Python reference library containing utility functions that are being used by other GitHub projects within the NF EA domain.

The historic NF architecture diagram below highlights the footprint of the NF EA COM (the main component of the NF EA Common Tools Library) along with the other libraries.



historic nf architecture - with library footprints

The code was developed using BORO Clean Coding Principles (see page 104).

> This GitHub project is currently closed, but may be sporadically updated by the BORO Development Team in the future.

(see page 218)

133 https://github.com/boro-alpha/nf_ea_common_tools

2.3.14 Training - bCLEARer - BORO Modelling Tutorial

One of the key skills for a good bCLEARer implementation, particularly at the EVOLVE stage is BORO modelling. BORO has its own approach to this, which was initially described in (Partridge 1996 Business Objects) (see page 140), and has subsequently evolved.

To assist with the acquisition of these skills, BORO has developed and evolved training modules, including a BORO Modelling Tutorial.

The aim of this BORO Modelling Tutorial is for the participant to develop an understanding of the BORO Foundational Ontology, and develop their ability to use the BORO Modelling Approach, within these modelling types:

- STM – Space-time Maps (including SDM – STM Domain Models)
- OED – Ontological Euler Diagrams
- BUML – BORO UML

This is a practical modelling course using a performance approach, so participants will perform tasks with model deliverables and then will assess their own performance through assessment and comparison of these deliverables.

(see page 219)

2.3.15 Training - bCLEARer - UNICLASS Example

BORO has developed training modules to aid people in adopting the bCLEARer approach[134]. One of these training modules is based upon an example implementation (UNICLASS) and takes the participant through the evolving stages of a bCLEARer project (see Evolutionary time (see page 8))

This UNICLASS example implementation code is stored in the UNICLASS bCLEARer Process (see page 208) GitHub repository. A description of the UNICLASS project can be found in (Partridge 2020 Implicit Requirements for Ontological Multi-Level Types in the UNICLASS Classification) (see page 147).

The main areas this training focuses on are:

- the bCLEARer approach[135]
- the BORO clean coding practices (see page 105) (see BORO Clean Coding Principles (see page 104))

BORO provided a small team of Shell employees this training during March and April 2022.

(see page 220)

134 https://borosolutions.net/bclearer-approach
135 https://borosolutions.net/bclearer-approach

2.4 Resources - Tables

2.4.1 bCFAP's nesting major decomposition levels

(Domain) thin slices level	the level where the scope is broken down into domains
bCLEARer stages level	the level that breaks the process over the domain into stages of the bCLEARer digital journey
bUnits level	the level that breaks each bCLEARer stage into base bUnit filters and their associated bUnit pipes

(see page 222)

2.4.2 bCFAP's architectural nesting levels - facet classification

level types	kind	modality	nestability
pipelines	core	mandatory	boundary (single)
thin slices	extension	optional	nestable
stages	core	mandatory	boundary (single)
sub-stages	extension	optional	nestable
bUnits	core	mandatory	boundary (single)

(see page 223)

2.4.3 bCFAP's architectural nesting level facets

Facet	Elements	Description
kind	core	mandatory core levels in the architecture that set the framework
	extension	optional extension levels that enable modularisation in the architecture
modality	mandatory	a mandatory level in the pipeline nesting architecture
	optional	an optional level in the pipeline nesting architecture
nestability	boundary slice	a single level - with no nesting - in the pipeline nesting architecture
	nestable	nesting is allowed - but not mandatory - in the pipeline nesting architecture

(see page 224)

2.4.4 bCFAP's core architectural nesting levels

levels	descriptions
pipeline	the top, root, level of nesting for the bCLEARer pipeline - there is only a single root pipeline filter at this level. It is the upper boundary of the pipeline architecture.
stage	a single boundary layer (with no nesting) that contains the filters for the stages of the bCLEARer digital journey.
bUnit	the leaf nodes in the nesting structure, the bUnit filters. As this is a hierarchical tree structure, they belong to one, and only one, bCLEARer stage. It is the lower boundary of the pipeline architecture.

(see page 225)

2.4.5 bCFAP's extended architectural nesting levels

levels	descriptions
thin slice	where there are a significant number of sequences of bCLEARer stages in a pipeline, inserting this level between the pipeline and the stages enables the stages to be grouped into a more hierarchical modular structure.
sub-stage	where a bCLEARer stage contains a significant number of bUnits, inserting this level between the stages and the bUNits enables the bUnits to be grouped into a more hierarchical modular structure.

(see page 226)