BUILDING FHIR INTERFACES
WITH TRANSFORMS BY EXAMPLE

A White Paper

Open Mapping Software Ltd

11 October 2018

Abstract:

FHIR is now the leading interchange language for connecting healthcare systems. There is a growing demand to build interfaces with transforms between healthcare systems and FHIR. This is expected to consume many thousands of man years of effort, and the costs may impede the adoption of FHIR.

Transforms By Example (TBX) is a new technique for building transform software. Using TBX to build FHIR interfaces can reduce the costs and accelerate the adoption of FHIR. This white paper discusses how TBX can be used in the most important emerging use cases. As well as reducing the cost of new transform development, TBX has two major benefits:

- Quality of transforms: the TBX process is requirement-driven, test-driven and can be done by a subject matter expert - giving higher quality than any process dominated by coding.
- Adapting and reusing transforms: the TBX process greatly reduces the cost of adapting a transform to new or local requirements

Table of Contents

1. THE NEED FOR FHIR INTERFACES AND TRANSFORMS

1.1 Adoption of FHIR and the Future of Healthcare IT

In the past few years, HL7 FHIR has become the dominant healthcare data interchange format, and is now chosen for nearly all new projects integrating information between healthcare IT systems. While legacy interfaces and exchanges between systems continue to be needed, for any new integration FHIR is now the preferred approach.

Better integration of healthcare IT systems is not a choice. It is essential, for many reasons:

- to control the exploding costs of healthcare
- to improve standards of patient care
- to exploit new opportunities for better care arising from medical science
- to exploit new opportunities from artificial intelligence
- to advance evidence-based medicine, and
- to increase the levels of patient involvement and control of their care.

All of these depend on making healthcare IT systems work together better than they do today. FHIR is now the key to that. No better data interchange standard will be available for many years.

So there is a large and growing demand to interface existing healthcare systems - and their exchange formats, such as HL7 Version 2 - with FHIR. This requires building data transforms between those data representations and FHIR.

Because if the huge number of healthcare applications requiring integration, and the large number of local variants of those applications, and the large number of FHIR resources and profiles, this will require a huge number of FHIR data transforms to be built and maintained.

While FHIR is without doubt the most cost-effective approach to integrating healthcare applications, deploying FHIR is going to be very expensive. The cost of building and maintaining FHIR interfaces will soon dwarf the costs of defining new FHIR applications, as is being done in forums such as HL7 International, for instance in published implementation guides. Building FHIR interfaces could become the dominant cost driver and roadblock for healthcare IT.

This white paper considers how those costs can best be reduced and controlled.

1.2 Emerging Use Cases for FHIR Integration

While we cannot anticipate what will be the main use cases for FHIR integration in even a few years hence, we can see the dominant use cases now and for the next two years. These are:

a) **Relational Databases to FHIR**: Probably more than 80% of all healthcare data is held in applications which use relational databases, and most of those will need to be FHIR-enabled. This requires developing relational-to-FHIR transforms in both directions - and converting various forms of FHIR search into efficient SQL queries on the databases.

b) **HL7 Version 2 to FHIR**: HL7 V2 is the most widely used healthcare data standard, working in hospitals all over the world for a range of applications - notably patient administration and laboratory

messaging. TheseV2 exchanges will not soon be replaced; FHIR needs to inter-work with them. This is not just a matter of developing a few standard V2-to-FHIR transforms. A particular challenge of HL7 V2 is its high level of local variation; it will be necessary to adapt any standard transforms to a large number of local variants of V2 (and profiles of FHIR), as efficiently as possible.

c) **FHIR Version to FHIR Version**: With remarkable rapidity, there has emerged a need to convert FHIR between versions - DSTU2, STU3, normative V4, and beyond. This is because anyone who builds a FHIR transform must target some version of FHIR. Different communities make this choice differently, and time forces change. Large parts of the market are already migrating between FHIR versions. This means migrating transforms between versions - and doing so for a wide range of resources and profiles. Efficient migration of FHIR transforms is a must-have.

d) **CDA to FHIR**: The market which is adopting FHIR most rapidly is the US market - which has already invested heavily in CCDA, for Meaningful Use. The vibrant US FHIR market needs CCDA to FHIR transforms. Other countries which have invested in CDA will soon have similar needs.

These are just four main groups of FHIR transform requirements. Each group on its own is a huge challenge, and there is then a 'long tail' of other transforms needed - to local special XML formats, and so on.

These examples show that the bulk of the challenge lies not in developing FHIR transforms in the first place - but later, in adapting them to local variants of the source, to changing requirements, and to different profiles, bundles, and versions of FHIR. This challenge will carry on for long after the first transform developers have moved on, when nobody wants to dig into their transform code. An efficient way to adapt transforms is essential.

1.3 Scale of the FHIR Integration Challenge

This section starts to estimate in ball-park terms how many transforms to and from FHIR will be required in the next few years.

Such an estimate could in principle be made by combining several different numbers, described below. That exercise is started here, but is not completed - because it is too difficult , and the resulting number is too large. What will actually get done will be less - the result of some worldwide design-to-cost exercise - building not all the FHIR interfaces that are required, but only the most urgently required of those interfaces.

The question is then: how much will the future integration and improvement of healthcare be held back, by our inability to build FHIR interfaces at an acceptable cost? What will be the human cost, in patient safety and patient care, if we fail to solve this challenge?

To understand the scale of the integration challenge, we need to compose (often by multiplication) a few numbers:

- **Different healthcare application types**: In a modern hospital, there are hundreds or thousands of different types of healthcare application. As medicine advances, the number grows, and most of these application types need integration with several other types of application, to exchange data.

- **Local variants of healthcare applications**: for each type of healthcare application, there are many different local markets (with different languages, different medical practices, and so on); and within each market, different commercial offerings and locally built solutions compete.

- **FHIR bundles, resources and profiles**: There are now over 140 FHIR resources, thousands of profiles on those resources, and bundles which can be composed of them. While any integration

problem may determine the broad pattern of bundles and resources chosen, there still remains a wide range of choices to be made.

- **Two-way transforms**: it is often necessary to transform data in both directions to interoperate.

- **Changing requirements for existing applications**: Any healthcare application will, during its lifetime, undergo many changes in its own intrinsic requirements; and perhaps more important, there will be changes in several other applications it must integrate with. Change in requirements at either end of an interface may drive change in the interface.

- **New applications**: These emerge not just from medical advances, but also from market experiment and adoption.

Estimating the worldwide requirement for FHIR transforms would require estimating each of these numbers, and then combining them. Often, the combination will be a multiplication. This is clearly a difficult sum to do - and equally obviously, the answer is a very large number. More FHIR integration is required than can be supplied. Whatever can be done to reduce the costs of individual transforms will help to bridge the gap.

1.4 Current Methods and Costs of Building Transforms

Currently, transforms between different representations of the same data are built in one of two ways: by using a mapping and transformation toolset, or by writing code in a procedural language.

Neither of these is satisfactory.

For general application integration, there is a large number of simple drag-and-drop mapping tools available. These have limited capability, and soon break down when faced with the complexity of healthcare data transforms. There are more sophisticated mapping tools, but they usually have a prohibitive learning curve, and even they have a sharply bounded capability. Their special-purpose languages and tools cannot compete with a mature general-purpose language like Java, for meeting the last 5% of awkward requirements.

So most healthcare data transforms are built by coding in a procedural language such as Java or C#. This has three drawbacks: cost, skills, and adaptability. The cost of coding and testing a typical healthcare data transform may run into tens of thousands of dollars, taking many months of elapsed time. The skills required to develop transform code are not the skills of a subject matter expert - so coders usually lack domain knowledge, and their code may meet the wrong requirements.

Then, when the transform is finally built and tested, key design decisions are buried in complex code, understood only by the developer (and then only for a limited time). When the developer moves on, or is reassigned, nobody dares touch the code, and it becomes un-maintainable. For even simple changes, or adaptations to meet a similar variant of the requirement, the usual choice is to start again. This is a massive waste of effort.

2. TRANSFORMS BY EXAMPLE

Transforms By Example (TBX) is a new technique for building complex data transforms, which can reduce the cost of writing transforms, while retaining the flexibility of coding in a high-level language, and simplifying the task of adapting transforms to new or varied requirements. The TBX toolset has been built to meet healthcare and FHIR requirements.

2.1 How TBX works

To build a transform with TBX, you do not make mappings or write code. You focus on your requirement - what the transform has to do.

You make a small number of **Example Pairs**, which illustrate what the transform is required to do. Each example pair is an instance of the source data, together with the target instance (such as a FHIR resource or bundle) which the transform is required to produce. Matching data values occur in the source instance and the target instance.

If the example pairs are accurate, the Transform By Example Toolset does the rest. These Eclipse-based tools:

- Generate the transform which converts the source to the target (and makes the inverse transform)
- Test the transform, giving it the source examples to create the target outputs.
- Compare the outputs with the supplied target examples, to summarise what is missing or incorrect - what the transform does not yet do correctly.
- Output the transform in deployable forms, such as Java code or FHIR Mapping Language.
- Display detailed diagnostics in the TBX Example Editor, to help refine the transform.

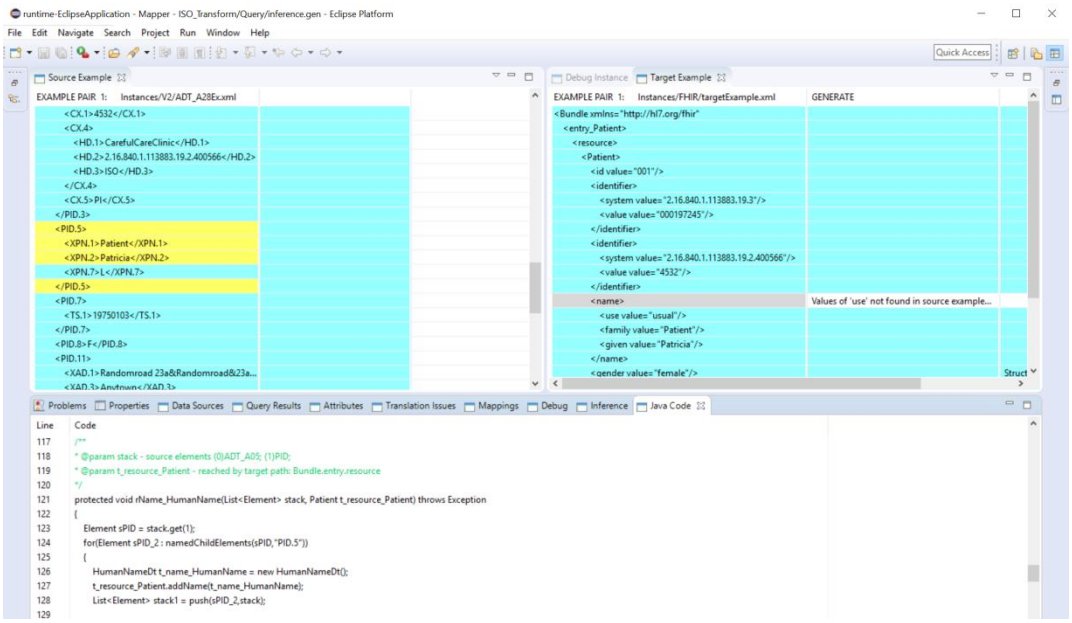
The TBX tools infer the required transform by supervised learning from examples. By spotting matching node numbers and matching data values in the source example and the target example, they reliably infer what mappings and transform are needed. In the few cases where your examples mislead the tools, or where they fail to spot some mapping, the problem is soon revealed by testing, and you are given guidance to refine the examples - or correct errors in them. You are the teacher, and TBX is the pupil.

Large numbers of example pairs are not required. An accurate transform can be generated from one good example pair, if that example embodies all the requirement. Creating good example pairs requires only an understanding of the requirement - and a few basic techniques to keep the examples as simple as possible.

As well as the examples which TBX uses to learn from, you can provide other test cases - which are not used to infer the transform, only to test it. All these tests are run automatically each time you refine the transform.

Most of the work of building a transform - the design, coding, and running test cases - is done automatically by the tools. The parts you have to do - providing the example pairs - are things you must do in any case, to understand the requirement and to test the transform.

The main activity required in using TBX is to edit the example pairs. You do this with a pair of intelligent editors, which show the source example and the target example side by side:



The example shown is a V2 to FHIR transform, after the transform has been automatically generated. The 'name' node in the FHIR target has been selected, so that the nodes in the V2 source which map to it are automatically highlighted in yellow. The pane at the bottom shows the generated Java code which creates the selected target node. Any problems found in transform generation are highlighted and can be examined.

The intelligent editors only allow you to create structures allowed by the source or target structure definitions, and they warn you of target data values that cannot be derived from the source. You tell the editor where fixed data values are required in the target. If code conversions or data format conversions are needed to get from source to target, you tell the tools what conversions are to be used.

Generally, the only knowledge you need to use the editors is knowledge of the requirement - not knowledge of any mapping or coding techniques. It can be done by a subject matter expert.

It is usually best to start with a simple source example, then build up the target example incrementally to match it, then add more complexity to the source. In this way the transform is built up incrementally in small steps, checking at each stage that it does what you require. We are exploring and recording a methodology to use the TBX tools in the easiest and most effective manner.

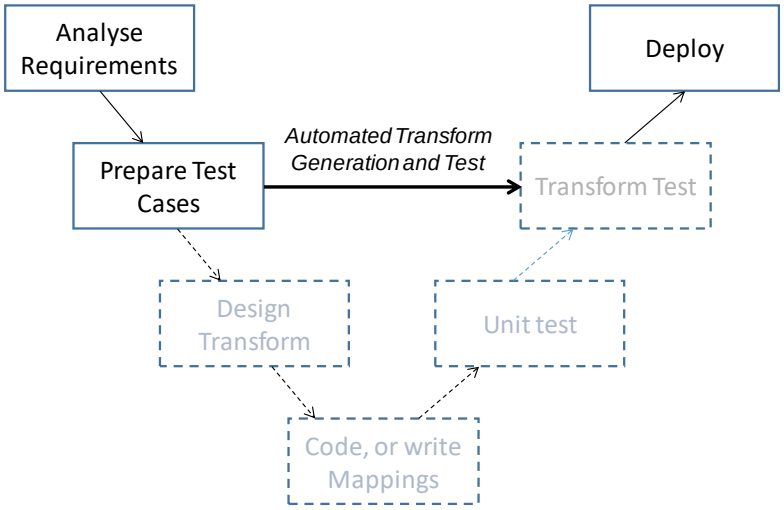
The edit-generate-test cycle of TBX is very rapid, leading to a highly productive form of test-driven development (TDD). Rather than getting involved in technical complexities of mapping or coding, you are brought face to face with the complexity and subtlety of the requirement. This can be disconcerting - but is much more productive.

2.2 Cutting the Costs of Transform Development

However you intend to develop a transform, you will to have to build test cases, to validate and test the transform when you have built it.

With TBX, defining the test cases is almost all you do, and the tools do the rest. Thus, TBX reduces the cost of building a transform, essentially as far as it can be reduced.

This can be seen in terms of the traditional 'V' model of the software development lifecycle. The V model is shown below, with dashed boxes showing the parts of the work which the TBX tools do automatically.



We do not yet have much calibration of the cost savings implied for developing a transform *ab initio* by TBX , but there are reasons to expect that the savings (compared to hand coding of transforms) are better than a factor two - doubling the productivity of a transform developer. A large amount of design, coding and debugging effort is simply bypassed.

2.3 Difficult requirements

Experienced developers of transform software know that there are always tricky requirements - caused, for instance, by data quality issues in the source - which cannot be dealt with in any automated, tool-driven way. Special code is needed to handle these requirements. That special code is best written in a general purpose high level language such as Java.

TBX can deliver any transform in generated Java code. When FHIR is the target, the Java makes calls to the HAPI reference implementation of FHIR. A small sample of the generated Java code for a V2 to FHIR transform is shown below.

```
/**
* @param stack - source elements (0)ADT_A05;
* @param t_resource_Patient - reached by target path: Bundle.entry.resource
*/
protected void pBirthDate(List<Element> stack, Patient t_resource_Patient) throws Exception
{
   Element sourceTop = stack.get(0);
   for(Element sPID : namedChildElements(sourceTop,"PID"))
   {
      List<Element> stack1 = push(sPID,stack);

      Node sBirthDate = namedChildNode(sPID,"PID.7");
      if (sBirthDate != null)
t_resource_Patient.setBirthDate(dateFormat.parse(FHIRConverters.date_V2_to_FHIR(null,getText(sBirthDate))));
   }
}
```

The generated Java follows a simple design pattern, which you can soon understand. It lets you see easily, for each Java method, where you are in the source and the target. The generated Java class inherits from a simple general-purpose superclass, which provides some of the methods used in the example above. Data conversion methods are supplied in another class (here FHIRConverters), and code conversions are supplied as FHIR ConceptMaps or csv tables. Other classes and methods come from HAPI.

A list of source elements (loosely called a 'stack' in the code - because the top element is often all that is needed) gives access from any method to all parts of the source instance which it may need, to create a part of the target.

Whenever there are special difficult requirements - or even where the TBX tools do not yet meet all the requirements - then you can modify the generated Java code to meet the requirement. If you need to do this, it is best to put the modified code in a subclass of the generated Java class, which overrides selected methods - so that you can re-generate the Java and not lose all your modifications.

For typical data format conversions between source and target (such date formats), or code conversions such as converting from LOINC to SNOMED, it is not necessary to override the generated code. You can provide code conversions to TBX in a FHIR ConceptMap resource, or in a csv conversion table. You provide data format conversions in pre-compiled Java methods (the code sample above includes a generated call to a pre-defined date format conversion method).

In typical transform cases, 95% of the Java code that you need is automatically generated by TBX - leaving you to focus on the last 5% of difficult requirements. You have the full flexibility of Java to meet those requirements. Your modified Java will be tested automatically within TBX, as soon as you compile it.

2.4 Improved Quality of Transforms

As well as the benefits in cost which come from automating much of the transform development lifecycle, TBX can deliver major improvements in the quality of the transform. These improvements arise for several reasons:

e) **Required skills and Domain Expertise**: When transform development is done in code by a developer, that developer is typically not a subject matter expert (SME). He is not an expert in the clinical domain, and the design decisions he makes when coding the transform are informed only by his own 'gloss' on the domain knowledge. These decisions may embody requirement errors, which are hidden in code and are not revealed by testing.

   With TBX, coding skills are not required to develop a transform; so the TBX tools can be driven by a subject-matter expert, who is aware of the subtleties in the requirement. Where hand coding is required for the last 5% of a transform, the SME can direct a developer, and make sure that the test cases fully exercise the requirement. As the most expensive software errors usually arise from a lack of domain knowledge, the ability of SMEs to use TBX is a vital driver of quality.

f) **Generated Code is Reliable and Repeatable**: Manual coding of a transform is subject to coding errors, which are sporadic and may be hard to detect in testing. Generated transform code is either all correct - or, if it is not correct, has errors in many applications, which are soon revealed and corrected. TBX has already passed though this stage, correctly generating a wide range of transforms.

g) **Emphasis on the Requirement**: When developing a transform in TBX, you do not spend your time on technical design and coding issues; your attention is brought straight to the requirement, in all its subtlety and complexity. This may at first be uncomfortable - having no familiar coding task to fill time to the coffee break - but crafting examples forces you to think about the requirement, and leads to better quality in the end.

h) **Test-Driven Development Style:** The best way to use TBX is to start simple - with small source and target examples, which illustrate a part of the requirement - and to build up to a full transform in small steps, editing the examples and testing all the way. The TBX tools let you do this in a rapid edit-generate-test cycle; taking less than a minute for a simple edit. This is Test-Driven Development (TDD), which has well known benefits in quality. It follows the FHIR ethos of learning by building and testing - which has helped FHIR to succeed better than previous healthcare standards.

For all these reasons, TBX can give much higher quality transform software than the usual hand coding of transforms. This saves effort and improves patient safety.

2.5 TBX Sources and Targets

TBX has been developed and tested to handle the following types of source and target:

- Profiled FHIR resources and bundles - JSON or XML
- Relational databases
- HL7 Version 2 - pipe-hat or XML
- CDA
- OpenEHR
- General XML, as defined by XML schema

Subject to some small limitations which will be removed, it is possible to have any of these types as source, and any as target.

2.6 Deployment options

TBX can deliver generated transforms in four different forms:

- A Java runtime engine, driven by declarative mappings
- Generated Java code
- Generated FHIR Mapping Language (FML)
- Generated XSLT

Each of these has its own advantages and disadvantages, as follows:

The Java runtime engine uses libraries from TBX and Eclipse. It can run transforms in both directions (source to target, and target to source) from the same TBX generation process. The declarative mappings are easy to understand in a graphical editor, and can be modified to meet some special requirements; but this is not as flexible as modifying the generated Java code.

The generated Java code runs faster than the runtime engine, and gives greater flexibility for manual modification than the runtime engine. It is generally the recommended deployment option. To generate inverse transforms, you need to transpose source and target examples in the TBX generation process.

Generated FHIR Mapping Language is more concise than Java code, and runs direct on HAPI with no compilation. Aside from the dependence on HAPI, it is platform-independent. It can be packaged as a FHIR StructureMap resource. It runs in only one direction from one generation process.

The generated XSLT runs fast and can run transforms in both directions, from one TBX generation process. Data value conversions and code conversions need to be expressed in XSLT.

2.7 FHIR Mapping Language

FHIR Mapping Language (FML), which can be expressed in the StructureMap resource, was introduced as a declarative, platform-independent way to define FHIR transforms. It is supported in the HAPI Java reference implementation, but not yet in the C# reference implementation. It is still at maturity level 0, and has relatively few users.

It limited uptake seems to be caused largely by the extra learning curve required for the unfamiliar declarative mapping language - which few people have mastered.

The TBX tools can generate a transform in FML automatically. They could also (although this has been rarely used, and is not fully working or tested) support a round trip between TBX declarative mappings and FML declarative mappings.

A sample of the FML mappings generated by TBX is shown below:

```
    rName_HumanName : for sNK1_1.NK1__2 as sNK1_3 make t_contact_Patient_Co.name as
t_name_HumanName_1  then {
      pFiller_4 : for sNK1_3.XPN__1 as sXPN  then {
        pFamily_1 : for sXPN.FN__1 as sFamily_1 make t_name_HumanName_1.family = sFamily_1
      }
      pUse_2 : for sNK1_3.XPN__7 as sUse_2 make t_name_HumanName_1.use =
translate(sUse_2,"#HumanName_use_conversion","code")
      rPrefix_string_1 : for sNK1_3.XPN__5 as sXPN_1 make t_name_HumanName_1.prefix = sXPN_1
      rGiven_string_1 : for sNK1_3.XPN__2 as sXPN_2 make t_name_HumanName_1.given = sXPN_2
    }
```

This is a small fragment of a simple V2 to FHIR transform. While generated FML is more compact than generated Java, its denseness and style make it harder to read. You can discern the V2.XML tag names in the variable names. The nested 'for' structures carry an implicit iteration in most implementations. For more complex transforms (such as OpenEHR to FHIR) the nesting is deep.

The TBX capabilities for FML might be used in two different ways:

- One can use TBX as an educational tool - to generate FML for simple or complex transforms, so users can see how FML works and learn the language
- One can use FML as a runtime engine for TBX transforms - regarding FML as a kind of machine code which users need not understand.

These are two very different styles of use, and it remains to be seen how they will be taken up. The jury is still out on FHIR Mapping Language, but TBX has facilities to support it.

2.8 Cutting the costs of maintenance and adaptation

TBX has economic and quality advantages when developing a new transform *ab initio*. It has even greater benefits when adapting some existing transform to a varied requirement.

Suppose for example that the new requirement differs from the previous requirement by about 10%. This means that 10% of the data elements in the source have changed their location, format, or meaning; and it is required to transform these new elements reliably to the target (whose structure may also have changed by 10%)

If, as is usually the case, the transform has been developed by hand coding, it is first necessary to understand the full structure of the existing transform code, in order to know what code needs to be modified. If the developer who wrote the transform has moved on or been reassigned, this is not an easy task. It may need both developer skills and subject matter expertise - not often found in one person. If the code is not of high quality, new code to meet 10% of new requirements will be scattered across more than 10% of the existing code - say as much as 40%. The ramifications of change, which all need to be tested, may stretch wider than that.

Typically this might mean that the cost of meeting 10% of new requirements is as much as 50% of the cost of developing the original transform. In such a case, NIH may win the day, in a decision: "Let's redo it all better from scratch" - leading to a 100% cost to meet a 10% change in requirements.

With TBX, adaptation to a new requirement is simpler:

a) Take some instances of the new, modified source. Run them through the existing transform, to create test cases (the targets will be 90% correct)

b) Correct the last 10% of the targets in the test cases, so they reflect the new requirement - what you intend the new transform to deliver.

c) Use some subset of these test cases as your example pairs, for TBX to infer the new transform

d) As required, refine the example pairs, until you get the transform you want.

Some things to note about this process:

- It is entirely focused on the requirement - so it can be done by an SME with deep domain knowledge, not by a coder.
- It requires no delving into existing transform code (except possibly the 5% of hand code as above, written to meet difficult requirements)
- It is entirely focused on the 10% of requirements which are new or changed
- Regression testing of the remaining 90% of the transform is automatically built into the process.
- It has all the benefits of improved quality noted above for *ab initio* development.

In summary, the cost of re-engineering a transform to meet 10% new requirements with TBX is likely to be 10% of the original cost, rather than 50%. As the original cost was at least 2 times smaller (because TBX saved the costs of design and coding), the overall saving can be as much as a factor of 10, and the quality better. This is a game-changer.

We anticipate that over time, the main cost of building FHIR transforms will lie not in the original development, but in the adaptation of transforms to local and varied requirements. So these benefits of TBX will be essential.

2.9 Reverse Engineering an Existing Transform

Many organisations are faced with the challenge of adapting existing transforms to new requirements (with or without FHIR); and, for reasons described in the previous section, they face high costs in doing so.

TBX offers those organisations a way out - to re-engineer their existing transforms into TBX form, so they can then be easily adapted to new requirements. The process for doing this is straightforward:

- Run your existing (non-TBX) transform, on a set of source examples - to produce a set of test cases
- Choose a subset of these test cases to serve as example pairs, for TBX to infer a transform from.
- Generate and test the TBX transform, using the example pairs to infer a transform, and testing with the other test cases.
- Then refine the transform to meet new requirement, as in the previous section.

This process can be applied regardless of the technology used to develop the original transform. It removes the need to understand and modify legacy transform code which 'nobody dares touch'. It preserves your investment in the legacy transform, and gives you a cost-effective way to meet new requirements.

2.10 A Market in Example Pairs

Data transforms are essential for healthcare interoperability, and are regarded as a valuable resource. Their value is often thought to reside in transform code or mappings.

However, with TBX, **the value resides in the example pairs** - not the transform code. This is because:

- Using TBX, a working transform can be generated from the example pair - and deployed in a variety of convenient forms; example pairs are portable and platform-independent.
- An example pair can be understood and reviewed by a SME - whereas code cannot
- It is much quicker and more reliable to adapt an example pair to new or local requirements, than it is to adapt transform code. Adapting an example pair can be done by an SME, not needing a coder.

While in the past, suppliers and providers may have regarded transform code as their valuable assets, in future they will regard their example pairs as the valuable asset - because for the main use case (adapting to new or varied requirements) example pairs are much more reusable and reliable.

So it is in the interests of suppliers and providers to protect their example pairs as a valuable asset, and to acquire other useful example pairs. It is in the interest of national healthcare authorities to seed and promote a market for example pairs - both proprietary and Open Source - in their countries.

3. STATUS AND FUTURE OF TBX

3.1 Current Status

The TBX tools are built on a mapping and transform design platform which has been used by Open Mapping Software Ltd over many years. The design tools run under Eclipse. Building the TBX 'learning layer' on top of these tools has taken approximately a year's effort. Without the underlying platform it would have taken much longer.

The TBX tools can handle the range of source and target types described above, and have been tested on them in several combinations. TBX is able to handle those applications, while of course not being 'complete'. Some transform requirements still require the user to write of hand code in Java, which could in future be generated automatically. There is a list of other possible enhancements.

TBX is currently in Beta test evaluation at about a dozen sites worldwide - including the US, Brazil, and Russia. The aims of the Beta test period are to:

- Reveal gaps in TBX functionality and important user requirements, to inform the development program
- Reveal gaps in user documentation and help, to be remedied
- Obtain feedback on the user interface and usability issues
- Define and create tutorial material
- Define a methodology for best use of the TBX tools
- Create reference implementations for important use cases - to be adapted to local variants
- Assess how much hand-written Java code is needed to meet difficult requirements (as a proportion of the generated Java code)
- Measure the productivity gains from TBX, both for *ab initio* transform development, and for adapting transforms to new requirements
- Understand real-world transform challenges in greater depth

OMS can support only a limited number of beta test sites without funding.

The TBX software is not currently Open Source, and a patent application is in progress.

Following the Beta test period, TBX will be licensed commercially, at modest annual per-user licence fees, which can be obtained from OMS Ltd on request.

3.2 Taking TBX Open Source

While TBX is currently a proprietary software product, our preferred long-term direction is to make it freely available, to maximise the benefits to the healthcare IT community worldwide, and to patients.

We do not expect this to happen in the short term. Before it can happen, the usefulness of TBX needs to be proved in projects. We also wish to validate the concept of example pairs as a reusable interoperability asset, supported by TBX, in the place of transform code.

If, as we expect, the community finds value in the exchange and availability of example pairs, with TBX as the platform, there would then be a case for making TBX freely available. This could happen given adequate support to compensate OMS for its investments, and to sponsor an ongoing Open Source TBX support and development project.

OMS's needs for financial compensation are modest. We believe that the benefits of TBX for worldwide healthcare, in supporting FHIR-based interoperability in this important respect, will repay the investment many times over.

HL7 could coordinate such a move by getting guaranteed subscriptions from its members to a TBX project. National healthcare organisations such as ONC and NHS Digital may wish to subscribe, to accelerate FHIR adoption in their countries. Healthcare IT suppliers could subscribe, as an alternative to individual commercial arrangements with OMS. The benefit of subscribing would be a seat at the table of the body that supervises the future support and enhancement of TBX. When sufficient guarantees of subscriptions are obtained to support OMS' compensation and the start of the Open Source support project, OMS would make over the IPR to HL7, which could then manage the Open Source project or create an organisation to do so. OMS would continue to be involved, facilitating skills transfer to the new project.

Before that, TBX will remain a proprietary product with modest licence fees - encouraging suppliers and providers to build up their libraries of example pairs. If this view of the future of transform development is correct, example pairs will be the principal source of value to the community.

4. USE CASE: V2 TO FHIR TRANSFORMS

Because of the widespread adoption and likely continued use of HL7 Version 2, it is essential to support V2-FHIR interworking. TBX has been used to develop V2 to FHIR transforms.

The Orders & Observations working group of HL7 is proposing a project to develop authoritative Version 2 to FHIR mappings, to support anyone who needs to create and maintain transforms. Initially the project will develop the data type mappings, and it will then move on to segment mappings.

At this stage and after, the project may publish reference mappings and transforms at segment and message level, for the most important V2 message flows such as patient administration and laboratory, using value set mappings from other working groups. Publishing transforms is the best test of the mapping work, and is in the FHIR spirit of 'build it and evaluate it'.

These transforms need to be developed with the least possible effort, and delivered in forms which are easy to deploy and adapt to local requirements. TBX has been proposed for this purpose, and OMS will contribute to the project. As above, the most reusable way to publish transforms would be as example pairs, but other forms can be published for users without TBX.

Developing the core transforms with TBX will require only subject matter expertise to develop and review the example pairs and test cases. Transforms can be published as example pairs, or in any of the forms output by TBX - as mappings for a Java runtime engine, as generated Java, as FHIR Mapping Language, or as XSLT.

A key part of the publication will be the example pairs and test cases, to enable any SME to review the transforms, to understand the capability of the published mappings and transforms. The example pairs will enable any organisation, using TBX, to adapt the transforms to its own local needs - including local variations in the use of V2, such as Z segments.

The process for doing this was described in the previous section. It is straightforward, and requires only SME knowledge. Briefly, any organisation can run its own V2 source examples through the published transform. This solves X% of the problem, where X might typically be 70, 80, or 90 percent. They can then refine these partial FHIR outputs of the transform to define what they want from the full transform. From these example pairs, TBX will generate and test a complete transform, to meet the local requirements.

This process is much more efficient than re-engineering transform code or mappings to meet local requirements, and requires only localised SME knowledge. In this way the project can supply reusable and adaptable V2-FHIR example pairs - as the best way to create the many local variant transforms that will be needed.

5. USE CASE: RELATIONAL DATABASES TO FHIR

Most healthcare IT applications use relational databases to store their data. Probably more than 80% of the healthcare data in the world is stored in relational databases.

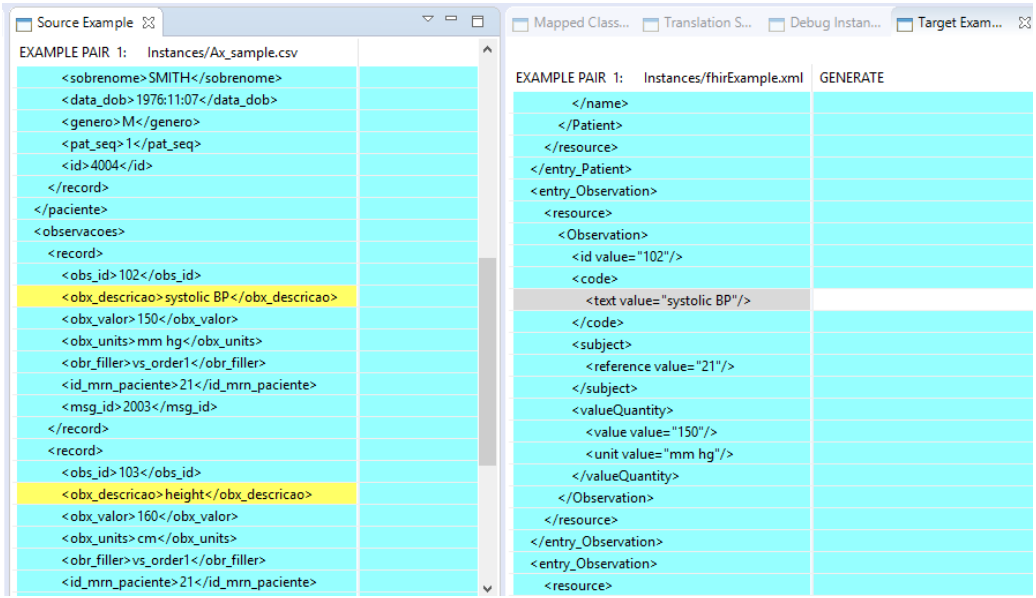
It is therefore essential to be able to expose and modify these data as FHIR - without migrating it to some specialised FHIR persistent form. That migration would either create duplicate copies of data with new synchronisation/replication problems, or render obsolete a vast body of application code which depends on SQL access to relational data.

So most IT application vendors now have a need to expose their relational data as FHIR, without re-hosting it. They are tackling this problem in a variety of ways - either by using a mapping approach of their own, or by hand-coding FHIR transforms. These methods have the drawbacks of cost, flexibility and maintainability described in previous sections.

TBX offers a more cost-effective way to solve the problem. TBX and its antecedents have been designed to handle relational data, and have been applied to relational data in a number of projects, including a FHIR server facade on a PAS system in a UK NHS Trust. The facilities in TBX for handling relational to FHIR transforms, while not yet complete, are complete enough to save costs and improve quality compared with conventional methods.

When building a Relational-FHIR transform with TBX, how do you create the necessary example pairs? You create the RDB source examples by running an SQL query against the database, to extract just those records needed to construct a FHIR bundle; and you convert the SQL output to comma-separated-value (csv) form. Then you create by hand the target examples - the FHIR resources or bundles - that you want to result from the transform.

When you give these example pairs to TBX, it shows the csv source input in a simple flat XML form, which can be edited field by field:



This example is taken from a simple database recording vital signs information, with table names and column names in Portuguese. As usual in TBX, you can select a target node to highlight nodes in the source mapped to it.

You edit the target example (and possibly the source example) to incrementally refine the transform, testing it at every stage. From the example pairs, TBX infers the transform. This includes inferring the necessary mapping conditions to reflect the prime key/foreign key constraints of the database.

TBX then has the capability from the inferred mappings:

a) to create a relational to FHIR transform, to retrieve FHIR bundles from the database

b) to generate HAPI Java to implement a HAPI server engine on the database

c) From any RESTful FHIR search string, to automatically generate the appropriate SQL query to efficiently retrieve just the required records from the database.

d) Using the native Java runtime engine, to run the transform in either direction.

Some facilities are not yet complete. For instance, the capacity to generate SQL queries has not yet been coupled to the HAPI server engine; and there is not yet a capability to generate Java or FML for a FHIR to relational transform. These gaps will be filled soon.

Even in its current state TBX offers the most cost-effective way to couple an existing relational database to FHIR; and even if you have already embarked on developing the transforms in some other way, you can easily reverse-engineer those transforms into TBX form, giving a more cost-effective way forward while protecting your previous investment.

6. USE CASE: FHIR VERSION MIGRATION

Many projects have started to interface healthcare data to one version of FHIR, and now find a need to interface it to another version.

One way to do this is to create (or use some supplied) transform between FHIR versions; and then to chain the transforms at runtime, as in X => FHIR DSTU2, followed by DSTU2 => STU3.

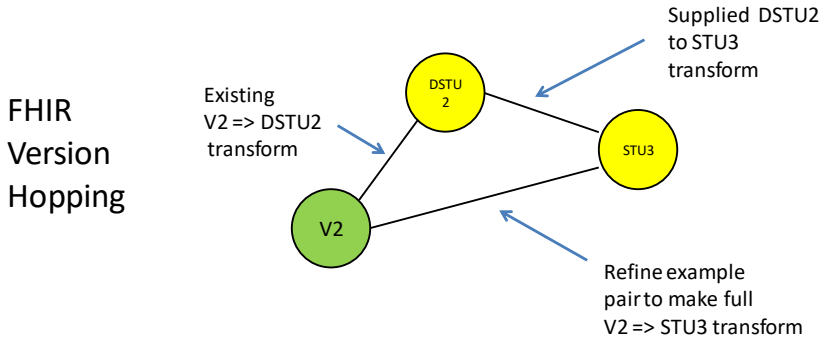
TBX has the capability to make a FHIR version to version transform (and many of these may be required, for profiled forms of each version). You use one FHIR version as the source example, and the other version as the target example. TBX is an efficient way to create these transforms, and then to adapt them to one of the many profiles in use, for the source or target version.

However, the approach of chaining transforms at runtime will necessarily build in the limitations of the intermediate version. As the X => DSTU2 transform must have the limitations of DSTU2, so must the final STU3 result from the chained transforms.

TBX allows you to do better - and to build a direct transform X => STU3 without any DSTU2 limitations. You do this as follows:

- From a source instance of X, run the existing X=> DSTU2 transform to create an instance of FHIR DSTU2
- Run a supplied DSTU2 => STU3 transform on this instance, to create a 'chained transform' FHIR STU3 instance - which has the limitations of DSTU2.
- Edit the STU3 instance so it contains all you want from the direct transform
- Use the source example X and the extended STU3 example as an example pair in TBX, to create the direct transform to FHIR STU3.

This process of 'version hopping' is illustrated below in the case of a V2 to FHIR transform:



Simple tricks like this will be needed to control the costs of maintaining transforms, in the face of proliferation of versions of both source and target. TBX allows you to do these things.

7. USE CASE: CDA TO FHIR

Because of the widespread adoption of CCDA in the US for Meaningful Use, there is a need for CCDA to FHIR transforms in both directions. Other countries will soon feel a similar need.

Some CCDA to FHIR transforms have been developed by Lantana Group in XSLT. In order to have them in a more maintainable and adaptable platform independent form, a project was started to convert these transforms to FHIR Mapping Language. That project is currently stalled for lack of effort.

The antecedents of TBX have extensive facilities for handling CDA and for using templates. These include a capability for simplified 'green CDA' which has been deployed on a small number of projects, and other CDA transforms developed in the UK.

The new TBX capability has not yet been applied to CDA to FHIR transforms. That application of TBX to CCDA is now being explored by Lantana Group. OMS Ltd are keeping a close watch on the work, and will adapt the facilities of TBX if necessary. Current indications are that little adaptation will be needed - but several alternative approaches are available if required.

This work may be used to seed the market for reusable CCDA-to-FHIR example pairs, for easy adaptation to local needs.

8. CONCLUSIONS

This paper has surveyed the challenge of building interfaces from existing healthcare IT systems to FHIR, as the best route to achieving greater interoperability. The conclusions are summarised here:

- The scale of the challenge of building FHIR interfaces is very large - because of the huge numbers of healthcare applications, local variants of them, and of FHIR resources and profiles. These numbers are multiplied in the costs.
- The costs of this may well delay the adoption of FHIR, and leave healthcare IT systems inadequately connected.
- Current techniques for building FHIR data transforms - using mapping tools, and writing transforms in code - have serious limitations of flexibility and cost.
- Transforms By Example (TBX) is a new technique for building transforms, which requires only creating source-target example pairs (test cases), to generate transforms automatically.

- TBX reduces the costs of developing transforms, as far as they can be reduced.
- Because TBX is requirement-driven and test-driven, and can be done by a subject matter expert, it delivers high quality transforms
- Because TBX generates Java code, it has the flexibility to meet difficult requirements
- The most important current use cases for FHIR transforms are HL7 V2 to FHIR, Relational Database to FHIR, and CDA to FHIR. All are required in both directions.
- TBX can greatly reduce the costs of adapting an existing transform to meet new or local requirements
- This makes TBX example pairs a valuable asset - more valuable and reusable than transform code
- The TBX tools are currently a proprietary Eclipse-based product, with modest licence fees
- We would like to take TBX Open Source, when the conditions are right.

These advantages imply that TBX is the future of healthcare data transform development. Help us build that future.