TTCN-3 User Conference 2009
Sophia Antipolis, France

Tutorial
Strategies in testing database application with TTCN-3

by Bernard Stepien, Liam Peyton, Grant Middleton

School of Information Technology and Engineering

Université d'Ottawa | University of Ottawa

uOttawa

L'Université canadienne
Canada's university

www.uOttawa.ca

Motivation on database testing

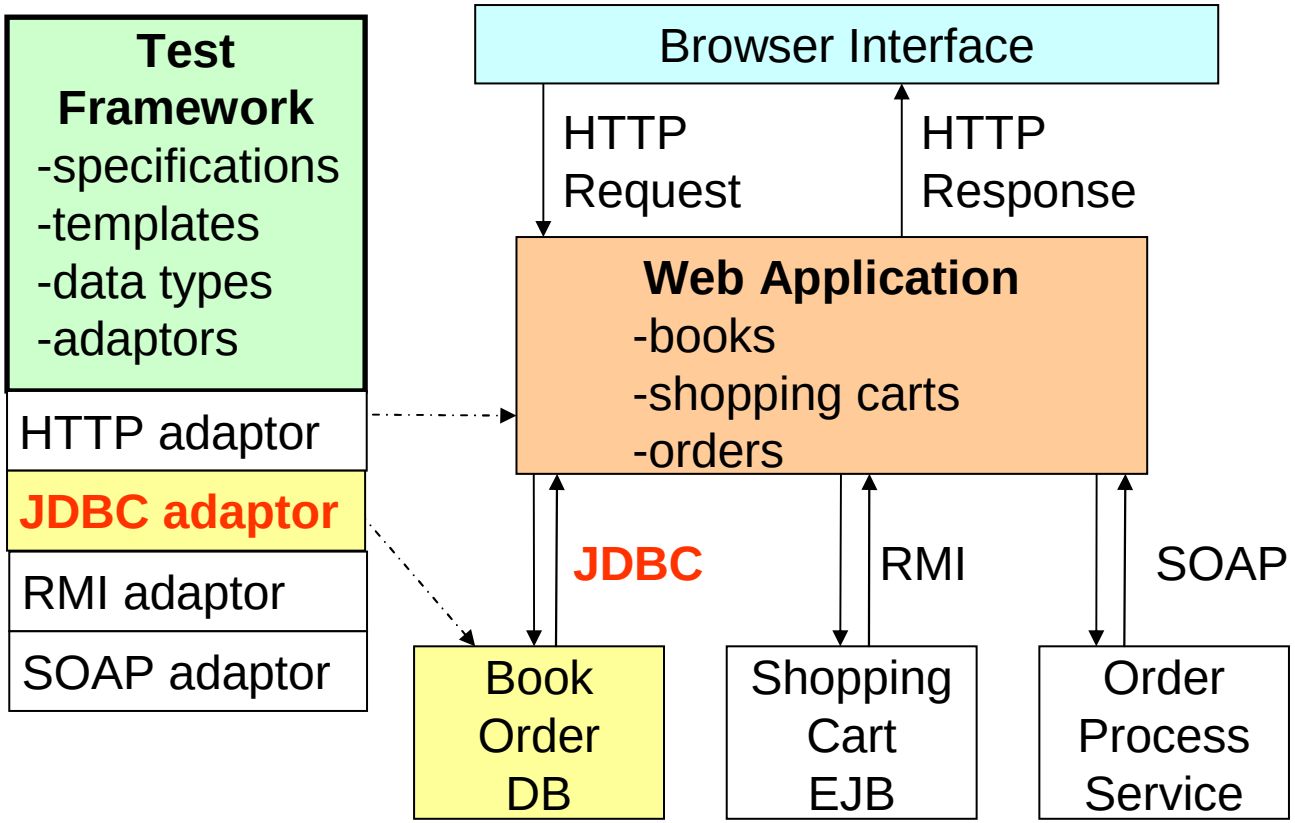
- Database testing is relatively trivial:
  - Send an SQL request
  - Check the results set
- Handling the results set at the codec level is also trivial but a repetitive task.
- For each different results set there has to be a different handling in the codec.
- Thus, normally, database testing is codec development intensive.
- We propose a solution that eliminates the intensive codec development effort.

Motivation on database applications testing

- Testing of database applications mostly consist in checking the database state following an operation in some application program.
- Thus, database application testing is a kind of integration testing where several individual test results need to be correlated.
- It is an advantage to be able to handle all aspects of integration testing with a single language like TTCN-3.
- Also, the various aspects can be correlated strictly at the abstract layer.

Service Oriented Architecture

Separation of concerns with concrete adaptors



Solution evaluation

single language/tool – TTCN-3

- The major contribution of this presentation is to demonstrate:
  - that a specification-based approach to integration testing enables one to define integration test campaigns
  - more succinctly and efficiently in a **single language/tool**
  - and correlate intermediate results in a **single data format**
- We evaluate the effectiveness of using TTCN-3 to support such an approach.

Testing a database state

SQL

- Creating tables
- Inserting data into tables
- Querying databases

Modeling a table, results set or SQL cursor in TTCN-3

- A Table row can be mapped to a TTCN-3 **record type**
- Table columns map to TTCN-3 **record type fields**.
- Sub-typing can be used to restrict sizes
- Table rows can be mapped to the TTCN-3 **record of** the basic row type.
- SQL NULL values can be mapped to the TTCN-3 omit value.

Abstract layer specification

- Simple shallow depth data types for update requests and query results sets

SQL

Field	Type	Null	Key	Default	Extra
Author	Varchar(30)	YES		NULL	
Title	Varchar(50)	YES		NULL	
price	Decimal(8,2)	YES		NULL	

TTCN-3

```
type record BookType {
   charstring author,
   charstring title,
   float price
}
```

template

```
template BookType amerique := {
      author := "Herge",
      title := "Tintin en Amerique",
      price := 8.20
}
```

Writing the codec

- There is a TTCN-3 standard called the TCI that provides classes and methods to achieve this.

TTCN-3 standard for codecs

Part 6: TTCN-3 Control Interfaces (TCI)

- Abstract Data Types (ADT) classes
- ADT manipulation Methods
  - Getters
  - Setters
- The standard does not provide any examples.
- Tool vendors provide limited examples
- Tool vendors provide class documentations

Myths about TTCN-3 codecs

- It is a common belief that writing TTCN-3 codecs is trivial.
- It is a very well known fact that writing codecs is one of the major hurdle for TTCN-3 adoption

Both of the above statements are myths

TTCN-3 standard for codecs

Abstract Data Value classes

Classes

- Value
- IntegerValue
- FloatValue
- BooleanValue
- ObjidValue
- CharstringValue
- UniversalCharstringValue
- BitstringValue
- OctetstringValue
- HexstringValue
- RecordValue
- RecordOfValue
- UnionValue
- EnumeratedValue
- VerdictValue
- AddressValue

TTCN-3 abstract layer types

- integer
- float
- boolean
- objid
- charstring
- universalcharstring
- bitstring
- octetstring
- hexstring
- record
- record of
- union
- enumerated
- verdict
- address

TTCN-3 standard for codecs

Abstract Data Types manipulation methods

CharstringValue:
- TString **getString**()
- void **setString**(in TString value)
- TChar **getChar**(in TInteger position)
- void **setChar**(in TInteger position)
- TInteger **getLength**()
- void **setLength**(in TInteger len)

RecordValue:
- Value **getField**(in TString fieldName)
- void **setField**(in TString fieldName, in Value value)
- TStringSeq **getFieldNames**()
- void **setFieldOmitted**(in TString fieldName)

TTCN-3 standard for codecs

Abstract Data Types manipulation methods

RecordOfValue:

Value **getField**(in TInteger position)
void **setField**(in TInteger position, in Value value)
void **appendField**(in Value value)
Type **getElementType**()
TInteger **getLength**()
void **setLength**(in TInteger len)

UnionValue:

Value **getVariant**(in TString variantName)
void **setVariant**(in TString variantName, in Value value)
TString **getPresentVariantName**()
TStringSeq **getVariantNames**()

■ ■ ■

Writing a codec for a query result

- A codec must use the expected abstract layer type name to switch to the appropriate decoder
- A TTCN3 type decoder processes a results set by setting values to named fields

```
public Value decode(TriMessage message, Type type) {
   if(type.getName().equals("BooksType")) {
      return decode_SelectBooksType(message, type);
   }
}
```

```
RecordValue bookValue = (RecordValue) bookType.newInstance();
String author = currentExec.resultsSet.getString("author");
CharstringValue authorValue = (CharstringValue)
                        bookValue.getField("author").getType().newInstance();
authorValue.setString(author);
bookValue.setField("author", authorValue);
```

codec considerations

Decode the results set from a SQL request

SQL Query

```
select * from books;
```

Results set

author	title	price
Herge	TINTIN et le temple du soleil	8.00
Herge	TINTIN et l ile noire	12.00
Herge	TINTIN en Amerique	8.20

Codec writing strategies

- Hard coding codecs
- Automated codec code generation
- Self-resolving codecs

Hard coded codec

general principles for decoding

- Build an instance of the appropriate abstract data class.
- Set values to those instances.
- This means setting:
  - Setting Single values to fields
  - Setting Complex values to Unions

Hard coded codec

decoder example

```
private RecordOfValue decode_BooksType(TriMessage message, Type type) {

  RecordOfValue booksValue = (RecordOfValue) type.newInstance();
  booksValue.setLength(0);
  Type bookType = booksValue.getElementType();

  try {
    while (currentExec.resultsSet.next() ) {
      RecordValue bookValue = (RecordValue) bookType.newInstance();
      String author = currentExec.resultsSet.getString("author");
      CharstringValue authorValue = (CharstringValue)
                                    bookValue.getField("author").getType().newInstance();
      authorValue.setString(author);
      bookValue.setField("author", authorValue);
      …// similar code for each field…
      booksValue.appendField(bookValue);
    }
    return booksValue;
  } catch (SQLException e) { … }
}
```

Drawbacks of hard coded codecs

Drawbacks

- A new decoder must be written for each abstract data type corresponding to a specific table or a specific results set.

Potential solutions

- An immediate solution would be to write a codec generator.
- Database results set decoding is one of the rare application area where a codec generator is feasible.
- TTCN-3 data types may also be generated automatically from the SQL results set.
- Generators can be simple for single tables results set but potentially complex for joined tables results sets.

results set and tables

- A Results set may have a different structure than the table it is derived from.
- A results set may only have a subset of the columns found in a table.
- A results set may span over several tables that are joined.
- A column in a results set may have been computed, thus is not found in the joined tables structures.

Codec design considerations

- Handling selected fields in a single table.
- Handling selected fields in joined tables.
- Convention: Ensure that the results set column names are the same as the TTCN-3 records field names.
- Clashes with TTCN-3 key words can be easily resolved by pre-pending the names with an underscore.

Single table field selection strategy

- Simple solution: make all fields optional

TTCN-3 type

```
type record BookType {
   charstring author optional,
   charstring title optional,
   float price optional
}
```

Template definition

```
template BookType amerique := {
      author := "Herge",
      title := "Tintin en Amerique",
      price := omit
}
```

SQL Query

Select **author, title** from books;

Results set

author	title	
Herge	TINTIN et le temple du soleil	
Herge	TINTIN et l ile noire	
Herge	TINTIN en Amerique	

Concept of a generic codec

- The number and nature of fields are unpredictable in a SQL select statement
- Solution:
  - Use the results set's **meta data** to discover the fields that are present.
  - Set the TTCN-3 abstract values only for the discovered fields

SQL selected fields results set

fields discovery

```
private RecordOfValue decode_SelectBooksType(TriMessage message, Type type) {

   RecordOfValue booksValue = (RecordOfValue) type.newInstance();
   booksValue.setLength(0);
   Type bookType = booksValue.getElementType();

   try {
      ResultSetMetaData meta = currentExec.resultsSet.getMetaData();
      int nbCol = meta.getColumnCount();

      while (currentExec.resultsSet.next() ) {
         RecordValue bookValue = (RecordValue) bookType.newInstance();

         for(int i=0; i < nbCol; i++) {
                        // handle each colum here …
         }
         booksValue.appendField(bookValue);
      }
      return booksValue;
} }
```

Handling a column

- Extract the column **name** and **type** from the results set meta data.
- Use the column **name** to retrieve the **data** from the results set.
- Use the column **name** and **type** to build the corresponding TTCN-3 **abstract value**.

Handling a column

```
String fieldName = meta.getColumnName(i+1);
String fieldTypeName = meta.getColumnTypeName(i+1);

if(fieldTypeName.equals("VARCHAR")) {
   String stringValue = currentExec.resultsSet.getString(fieldName);
   CharstringValue fieldValue = (CharstringValue)
                              bookValue.getField(fieldName).getType().newInstance();
   fieldValue.setString(stringValue);
   bookValue.setField(fieldName, fieldValue);
}
else if(fieldTypeName.equals("DECIMAL")) {
   Double floatValue = currentExec.resultsSet.getDouble(fieldName);
   FloatValue fieldValue = (FloatValue)
                             bookValue.getField(fieldName).getType().newInstance();
   fieldValue.setFloat(floatValue.floatValue());
   bookValue.setField(fieldName, fieldValue);
}
else
   System.out.println("unhandled field type name: "+fieldTypeName);
```

What progress have we made?

- We have eliminated the hard coding of field data manipulations.
- We need to set the unselected field to the value **omit** in the templates. (un-elegant)
- We have not eliminated the dependency between a TTCN-3 abstract data type and a specific decoder in the codec.

```
public Value decode(TriMessage message, Type type) {

   if(type.getName().equals("BooksType")) {
      return decode_SelectBooksType(message, type);
   }
   else …
}
```

Drawbacks of direct abstract types to codec mapping

- You still have to write a modify the codecs decode method to be able to handle different results set.
- this is particularly annoying when handling results set from SQL joins.
- However this decoder can now use a generic data extraction method.

Hard facts

- You possibly can not avoid defining different abstract data types for each results set.
- But you could entirely avoid writing the corresponding codecs.

Dilema

- Can we eliminate results set typing at the abstract layer?
- **No**, because for TTCN-3 test oracles, the templates are type based.
- Can we eliminate the remainder of hard coding the decoders corresponding to the abstract results set types?
- Yes. This is the purpose of this tutorial.

Key design consideration

- There is no equivalent to the SQL join to programming languages type or class system for merging types or classes definitions, neither for:
  - General purpose languages (GPL)
  - TTCN-3
- Even a join of types would not handle computed fields.
- Consequence:
  - we can not avoid the TTCN-3 abstract data type specification phase.
  - We must specify individual types for joined tables selected and/or computed fields.

Principles of a universal results set codec

- Specify a separate data type for each results set
- Combine all data types in a TTCN-3 union type.
- Thus the codec will have to detect only a single union type with a predictable name.
- Then handle union variants with a generic codec
- This codec is then a natural framework.

Advantages of the union type

- it completely relieves the tester from:
  - Codec writing
  - Codec generation
  - Codec modifications

Abstract typing strategy

user defined types

Basic tables types

```
type record BookType {
   charstring author,
   charstring title,
   float price
}

type set of BookType BooksType;
```

```
type record OrderType {
   charstring title,
   integer quantity
}

type set of OrderType OrdersType;
```

Join and computed field results set type

```
type record InvoiceLineType {
   charstring author,
   charstring title,
   float price,
   integer quantity,
   float amount
}

type set of InvoiceLineType
                              InvoicesLineType;
```

Field amount is a computed field

Abstract typing strategy

framework interface types

Interface type:

```
type union ItemType {

}

type set of ItemType ItemsType;
```

User implementation

```
type union ItemType {
   BookType booksTable,
   OrderType ordersTable,
   InvoiceLineType invoiceLineResult
}
```

- The union type **ItemType** is similar to an OO interface.
- The user implements the interface by merely filling the union type with the appropriate user defined types.

Template definition example

```
template ItemType ameriqueLine := {
   invoiceLineResult := {
     author := "Herge",
      title := "TINTIN en Amerique",
      price := 8.20,
      quantity := 10,
      amount := 82.00
   }
}
…
```

- Merely add the union variant specification to the type used.

```
template ItemsType t_invoiceLines := {
   ameriqueLine, temple_soleilLine, ile_noireLine};
```

Test execution

Existing tables

Books table

author	title	price
Herge	TINTIN et le temple du soleil	8.00
Herge	TINTIN et l ile noire	12.00
Herge	TINTIN en Amerique	8.20

Orders table

title	quantity
TINTIN et le temple du soleil	10
TINTIN et l ile noire	25
TINTIN en Amerique	3

SQL query

```
select author, books.title, price, quantity, price*quantity as amount
                from books inner join orders on books.title = orders.title;
```

Results set

author	**title**	**price**	**quantity**	**amount**
Herge	TINTIN en Amerique	8.20	10	82.00
Herge	TINTIN et le temple du soleil	8.00	25	200.00
Herge	TINTIN et l ile noire	12.00	3	36.00

SQL Join test case specification

- All test cases use only one type for the receive statements

```
testcase TC_HergeJoinSelect() runs on MTCType system SystemType {

   map(mtc:dbPort, system:system_dbPort);

   dbPort.send(" select author, books.title, price, quantity,
                        price*quantity as amount from books
                        inner join orders on books.title = orders.title ");

   alt {
      [] dbPort.receive(t_invoiceLines) {  // ItemsType
         setverdict(pass);
      }
      [] dbPort.receive(ItemsType:?) {
           setverdict(fail);
      }
    }
}
```

Union type codec development Strategy

Codec decode() content

- Contains only one type to switch on, namely ItemsType

```
public Value decode(TriMessage message, Type type) {

   if(type.getName().equals("ItemsType")) {
      return decode_ItemsType(message, type);
   }
   else
      System.out.println("in decode "+type.getName()+" not implemented");

   return null;
}
```

Union type decoder structure

- Since we are handling a union type, the key to the decoder is to know the union **variant name**.
- Since a Union type specifies that any of the variant could appear and that in the database case, only one kind of variant can be present, it is essential to have the correct variant name in order to be able to construct the abstract value.
- There is **no provision** in TTCN-3 to indicate a unique variant kind to the decoder.
- The only solution is to send the variant name in the request at the TTCN-3 send level.

Variant indicator in DB request

- Create a special request type that contains both the variant name of the expected result and the SQL query itself.
- Store that variant name in the test adapter and transmit it to the codec so that it can use it during the decoding of the results set.

```
type record DBSelectRequestType {
    charstring sqlQuery,
    charstring resultVariant
}
```

```
template DBSelectRequestType t_join_query := {
  sqlQuery := "select author, books.title, price, quantity, price*quantity as amount
                                from books inner join orders on books.title = orders.title",
  resultVariant := "invoiceLineResult"
}
```

Processing the abstract data type

1. Obtain the element type
2. Obtain the list of union variants names
3. Obtain the appropriate abstract value type using the variant name.

```
private RecordOfValue decode_ItemsType(TriMessage message, Type type) {
   RecordOfValue itemsValue = (RecordOfValue) type.newInstance();
   itemsValue.setLength(0);
   Type ItemType = itemsValue.getElementType();          (1)
   Value theElementValue = ItemType.newInstance();

   String VariantNames[] = theElementValue.getVariantNames();   (2)
   String theVariantName = "";

   for(int v=0; v < VariantNames.length; v++) {
      if(currentExec.getResultsSetName().equals(VariantNames[v])) {
           theVariantName = VariantNames[v];                    (3)
           break;
      }
   }
```

Processing the results set

1. Obtain the abstract value for a DB row
2. Obtain the list of field names for the record type
3. Process the results set and construct the returned abstract rows

```
RecordValue theRecordValue = (RecordValue)                    1
                                    theElementValue.getVariant(theVariantName);

String theUnionFieldNames[] = theRecordValue.getFieldNames();2
int nu = theUnionFieldNames.length;

while (currentExec.resultsSet.next() ) {     3
  UnionValue theReturnUnionValue = (UnionValue)
                                    theElementValue.getType().newInstance();

  RecordValue itemValue = (RecordValue)
                                    theRecordValue.getType().newInstance();
  …
```

Field processing

1. For each field, get its abstract value representation.
2. For each field type, extract the results set value using the corresponding JDBC method.
3. Set the TTCN-3 abstract value to the obtained database value.

```
for(int j=0; j < nu; j++) {
   Value fieldValue = theRecordValue.getField(theUnionFieldNames[j]);     1

   if(fieldValue.getType().getName().equals("charstring")) {      2
      String dbValue = currentExec.resultsSet.getString(theUnionFieldNames[j]);
      CharstringValue theCharstringValue = (CharstringValue)
                                                   charstringType.newInstance();
      theCharstringValue.setString(dbValue);
      itemValue.setField(theUnionFieldNames[j], theCharstringValue);      3
}
```

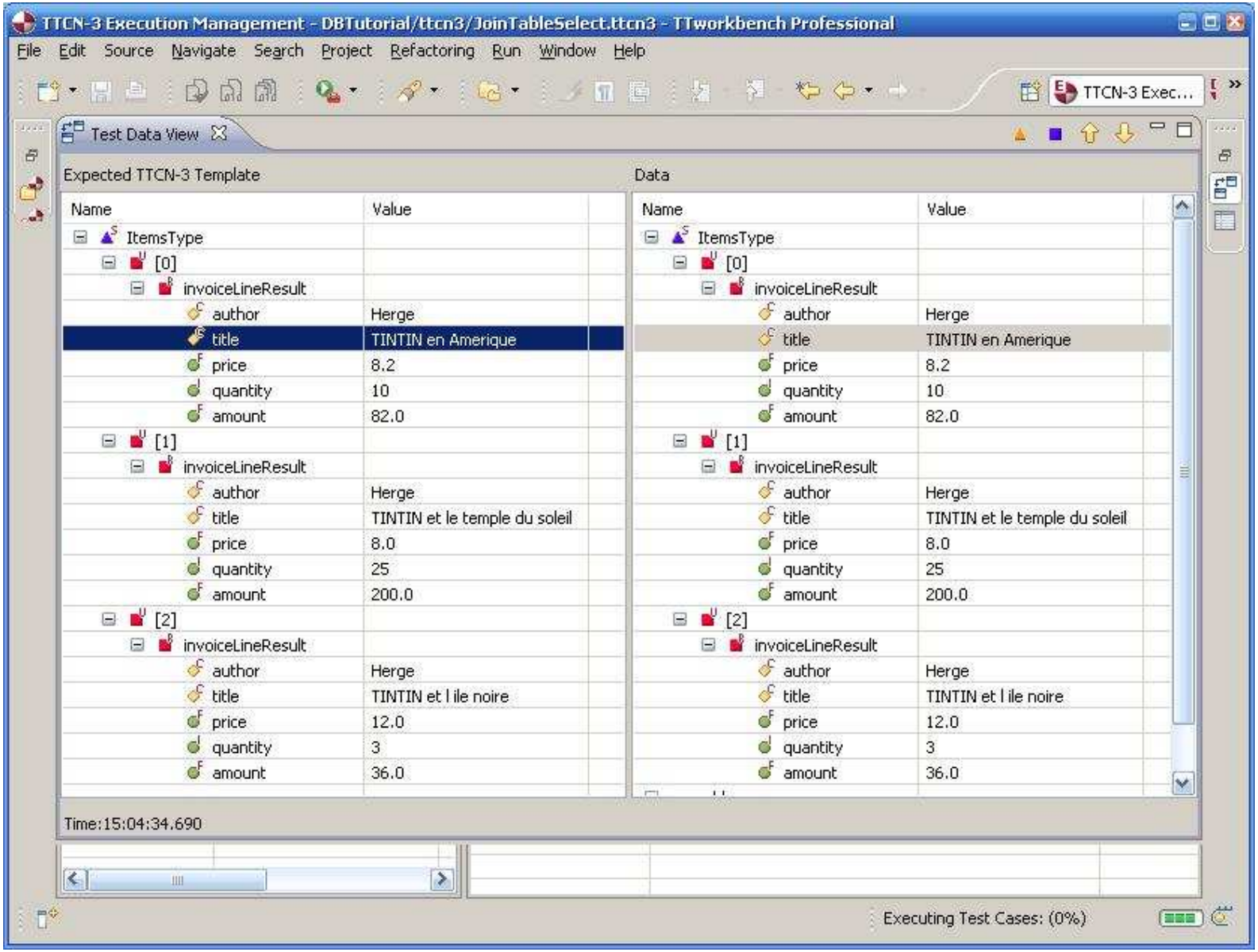
Some additional strategies

- Which side shall we use to discover field names and field types?
  - The results set meta data?
  - The TTCN-3 abstract data types?
- When considering the union type, we can only start with the TTCN-3 abstract data type.

Passing the union variant?

- Can we avoid passing union variants?
- Only in limited cases:
  - When the results set is from a single SQL table.
  - When the results set is a cursor.
- Thus, for reasons of generality, the passing of the variant name is mandatory.

Matching results



Automated code generation

- We have shown that there is no longer any need for generating a codec for database results sets.
- There could still be a need for generating the TTCN-3 type declarations for data base results sets.

Phil Zoio's database testing practices comparison with TTCN-3

published in Oracle Magazine, 2005

Zoio's recommendation TTCN-3 enforcement

- Zoio has clearly identified common problem areas in data base testing and made recommendations.
- But the recommendations implementation is left to the discretion of the developer.
- There is an interesting mapping between Zoio's recommendations and TTCN-3 language constructs.
- With TTCN-3, there is a model that is strictly enforced by the compiler.

Zoio’s testing best practices

- Practice 1: Start with a "testable" application architecture.
- Practice 2: Use precise assertions.
- Practice 3: Externalize assertion data.
- Practice 4: Write comprehensive tests.
- Practice 5: Create a stable, meaningful test data set.
- Practice 6: Create a dedicated test library.
- Practice 7: Isolate tests effectively.
- Practice 8: Partition your test suite.
- Practice 9: Use an appropriate framework, such as DbUnit, to facilitate the process.

Practice 2: Use precise assertions

Zoio

```
public void testPreciselyListPersons() {

        List results = dao.listPersons("Phil", new Integer(25));
        assertNotNull(results);
        assertEquals(2, results.size());

        for (Iterator iter = results.iterator(); iter.hasNext();)  {
                Person person = (Person) iter.next();
                assertNotNull(person.getId());
                assertEquals("Phil", person.getFirstName());
                assertNotNull(person.getAge());
                assertTrue(person.getAge().intValue() >= 25);
                assertNull(person.getSurName());
                assertNull(person.getGender());
        }
}
```

Practice 2: Use precise assertions

TTCN-3

- The TTCN-3 template concept is a natural answer to this problem.
- Zoio's example is mostly about checking that individual fields are not null
- How about checking real values?
- Checking real values using Java may require some additional manipulations.

```
type record PersonType
  charstring Id,
  charstring firstname,
  integer age,
  charstring surname,
  charstring gender
}
```

```
template PersonType t_phil :{
  Id := "A1234",
  firstname := Phil,
  age := (24..999),
  surname := "Doe",
  gender := "M"
}
```

Practice 3: Externalize assertion data - Zoio

- Place your assertion data in an **external repository**, to make your tests easier to manage and maintain.
- Most developers agree that writing precise assertions is a great idea but might not like the way one of our precise assertions—assertEquals(2, results.size())—is written, because the assertion value is **hard-coded**.
- If you're testing a large application with hundreds or even thousands of tests, you certainly don't want hundreds or thousands of hard-coded String or int values scattered throughout your test code, for two reasons.
  - First, if your test data changes, you want to be able to **easily find the assertion data** that needs to change.
  - Second, you will want to **take advantage of mechanisms for sharing assertion data across different tests**. The solution to the problem is to externalize your assertion data, as you would externalize String messages in your production code.

Practice 3: Externalize assertion data – TTCN-3

- Simple TTCN-3 solutions:
  - the TTCN-3 template because of its natural reusability.
  - Place templates in a separate module

Practice 7: Isolate tests effectively

- Zoio: Isolate tests so that one test's errors of failures don't affect other tests or prevent them from executing successfully.
- This is achieved easily in TTCN-3 due to the capabilities of abstraction.

```
Control {
   execute(myTest_1);
   execute(myTest_2);

   execute(myTest_3);
}
```

Practice 8: Partition your test suite

TTCN-3

- Concept of test case
- Re-usability of templates
- Concept of TTCN-3 functions to encapsulate fragments of behavior.

Reusing TTCN-3 Templates

- The template is also a very powerful structuring concept since it can be re-used by other templates
- a list of database items defined individually as above can be re-used to define a list of items
- Finally the previous template can be redefined in a more concise way

```
template CatalogEntry amerique := {
   booksInfo := {
      author := "Herge",
      title := "Tintin en Amerique",
      price := 8.20
   }
}
```

```
template ItemsType myBooks :=
      { templeSoleil, ileNoire, amerique }
```

```
template DBSelectResponseType myBooksSelectResponse := {
   result := "booksInfo",
   items := myBooks
}
```

Testing data base applications

Integration testing and databases

- Web applications
- SOA applications
- Legacy applications that produce some output, like a report or very concrete results such as a list of paychecks, etc…

Web applications

- Web applications in a service oriented architecture may have to integrate data from several data sources
- One aim of Integration testing is to verify intermediate results at key interaction points within the architecture of the web application
- This can be done by testing the database state.

Approaches

- Traditional approaches to integration testing typically use a variety of different test tools (such as HTTPUnit, Junit, DBUnit) and manage data in a variety of formats (HTML, Java, SQL)
- This is done in order to verify web application state at different points in the architecture of a web application
- Managing test campaigns across these different tools and correlating intermediate results is a difficult problem

Motivation

- The major contribution of this presentation is to demonstrate that a specification-based approach to integration testing enables one to define integration test campaigns more succinctly and efficiently in a single language/tool and correlate intermediate results in a single data format
- We evaluate the effectiveness of using TTCN-3 to support such an approach.

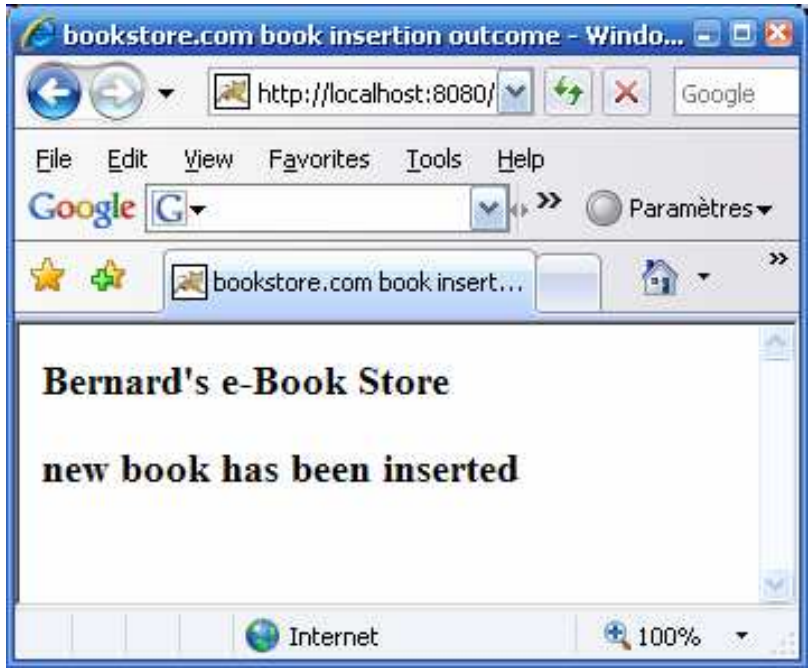
Integrating Data base and web application testing

Use cases

- Use case 1:
  - Use enters information through web forms
  - Check if the data base reflects the web forms entered information
- Use case 2:
  - Populate a database with information
  - User performs a query via a web form
  - Verify if the web response contains the same information as the database.

Web data entry test

Inserting information into database



Modeling web forms and results pages

- Web data most of the time doesn’t contain information about its nature.
- Web pages are focused on presentation.

Essential abstract layer elements

- Specify an html form

```
type record ParameterValuesType {
   charstring parmName,
   charstring parmValue
}

type set of ParameterValuesType ParameterValuesSetType;

type record FormSubmitType {
   charstring formName,
   charstring buttonName,
   charstring actionValue,
   ParameterValuesSetType parameterValues
}
```

Web Form Submissions as templates

```
template ParameterValuesSetType filledFormAmerique := {
   {parmName := "author", parmValue := "Herge"},
   {parmName := "title", parmValue := "TINTIN en amerique"},
   {parmName := "price", parmValue := "8.00"}
}
```

```
template FormSubmitType webInsertionFormSubmit
                                (ParameterValuesSetType theParameters) := {
   formName := "bookAdditionForm",
   buttonName := "add",
   actionValue := "http://localhost:8080/eBookStore/servlet/book_insertion",
   parameterValues := theParameters
}
```

```
web_port.send(webInsertionFormSubmit(filledFormAmerique));
web_port.receive(webResponsePage);
```

Web data entry test case

- Define filled forms templates
- Submit the filled forms
- Prepare the corresponding data base results set oracle

Web data entry test case

templates preparation

```
testcase web2DatabaseResultsTest() runs on MTCType system SystemType {
   var DBSelectResponseType theDBSelectResponse;

   map(mtc:dbPort, system:system_dbPort);
   map(mtc:webPort, system:system_webPort);

   // database re-initialization
   dbPort.send("delete from books");
```

```
   // have a user insert a book through a web page form
   webPort.send(webInsertionFormSubmit(filledFormOrNoir));
   webPort.send(webInsertionFormSubmit(filledFormAmerique));
```

Specifying the test oracle template

- Two different strategies
  - Hard coded template
  - Computed template
- Computed templates are more flexible
- Computed templates save coding
  - Small initial investment
  - Savings are revealed with economies of scale.

Results set hard coded template

```
template DBDeleteResponseType expectedDatabaseResults := {
   result := "booksTable",
   items := {
         {
            booksTable := {
               author := "Herge",
               title := "TINTIN au pays de l or noir",
               price := 6.0
            }
         },
         {
             booksTable := {
                author := "Herge",
                title := "TINTIN en Amerique",
                price := 8.0
             }
          }
      }
}
```

Computed results set template

- Transform the list of filled forms into database records.
- Match the above transformation results to a database query

Obtain template via transformation

- transform the list of filled forms information into a database query results template

```
var ItemsType expectedDatabaseResults :=
                transformForms2DB({filledFormOrNoir, filledFormAmerique});
```

Performing the test

- Use the generic type for data base communication that uses the union type.
- Pass the computed list of books as a parameter

```
// check if the database contains the entered books

dbPort.send(myBooksSelectRequest);
alt {
   [] dbPort.receive(myBooksSelectResponse(expectedDatabaseResults))
                                                    -> value theDBSelectResponse {
         setverdict(pass)
   }
   [] dbPort.receive {  setverdict(inconc); stop
   }
}
```

Web form transformation details

- For each set of form parameters, extract the parameter values using the field names as a key.
- Once all fields extracted, use them to build a database row.

Transforming web information into data base information

```
function transformForms2DB(FormsParametersValuesSetType theFormParms)
                                                    return ItemsType {
   var ItemsType theItems := {};
   var integer numOfFormParms := sizeof(theFormParms);
   var integer i;
   var ItemType anItem;

   for(i:=0; i < numOfFormParms; i:=i+1) {
      anItem.booksTable.author := getFieldValue("author", theFormParms[i]);
      anItem.booksTable.title := getFieldValue("title", theFormParms[i]);
      anItem.booksTable.price := str2float(getFieldValue("price", theFormParms[i]));

      theItems[i] := anItem;
   }

   return theItems;
}
```

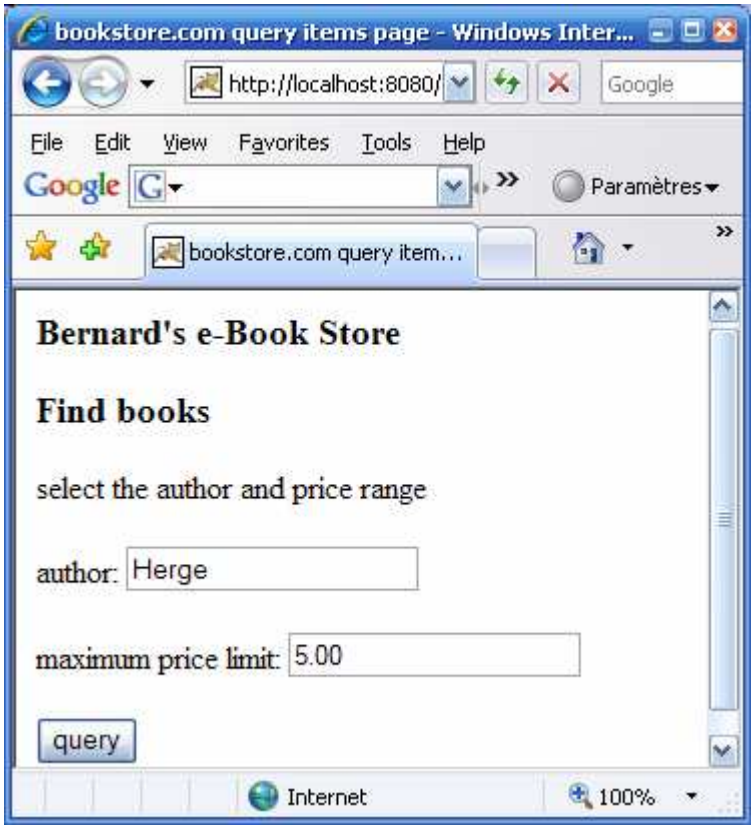
Extracting a field value from a form

- Searches the list of form parameters
- Matches on a parameter name
- Returns the corresponding value

```
function getFieldValue(charstring fieldName, ParameterValuesSetType theParameters)
                                                          return charstring {
  for(i:=0; i < numOfParms; i:=i+1) {
    aParameter := theParameters[i];
    if(aParameter.parmName == fieldName) {
      return aParameter.parmValue;
    …
```

Web query application test

Modeling a web query




```
template FormSubmitType queryBooksHerge := {
   formName := "queryForm",
   buttonName := "query",
   actionValue :=
        "http://localhost:8080/eBookStore/servlet/...",
   parameterValues := {
     {parmName := "author", parmValue := "Herge"},
     {parmName := "maxPrice", parmValue :="10.0"}
   }
}
```

webPort.send(**queryBooksHerge**);

↓

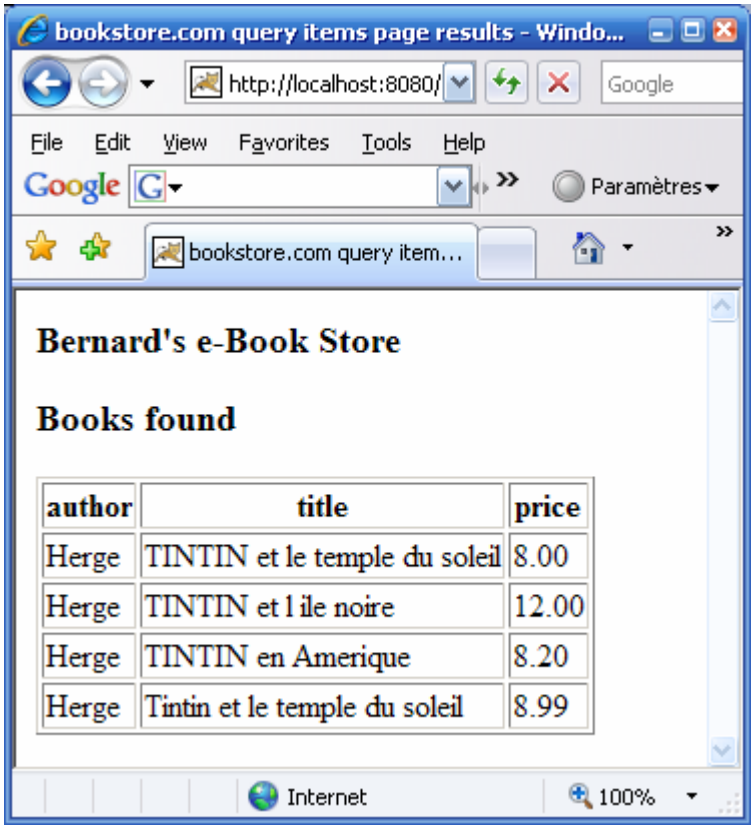
codec

↓

http://localhost:8080/eBookStore/servlet/book_selection?&author=Herge&maxPrice=10.0

Modeling a Web Response

abstract data types




```
type record WebPageType {
   integer statusCode,
   charstring title,
   charstring content,
   LinkListType links optional,
   FormSetType forms optional,
   TableSetType tables optional
}
```

```
type set of charstring RowCellSetType;
type record TableRowType {
   RowCellSetType cells
}
type set of TableRowType TableRowSetType;
type record TableType {
   TableRowSetType rows
}
type set of TableType TableSetType;
```

Hard coded test oracle

```
template WebPageType t_herge_query_response := {
   statusCode := 200,
   title := "bookstore.com query items page results",
   content := pattern "<HTML>*Bernard's e-Book Store*</HTML>",
   links := {},
   forms := {},
   tables := {
     {
        rows := {
           {cells := {"author", "title", "price"}},
           {cells := { "Herge", "TINTIN et le temple du soleil", "8.00"}},
           {cells := {"Herge", "TINTIN et l ile noire", "12.00"}},
           {cells := {"Herge", "TINTIN en Amerique", "8.20"}}
        }
     }
   }
}
```

TTCN-3 integration test description

- Perform a data base query and obtain the results set.
- Transform the results into the web response test oracle
- Perform the web query via TTCN-3
- Match the response with the database result set test oracle

Parametric test oracle

- The computed tables are passed in as a parameter

```
template WebPageType hergeDBQueryResultsPage(
                                        template TablesType theTables) := {
   statusCode := 200,
   title := "bookstore.com query items page results",
   content := pattern "<HTML>*Bernard's e-Book Store*</HTML>",
   links := {},
   forms := {},
   tables := theTables
}
```

Web test oracle as a transformation of database templates

```
dbPort.send(myBooksSelectRequest); // "where author = 'Herge'"
dbPort.receive(DBSelectResponseType:?) -> value theDBSelectResponse;
var ItemsType theReceiveDBItems := theDBSelectResponse.items;
```

```
var TableSetType booksTables :=
                    transformDBResultsIntoHTMLTables(theReceiveDBItems);
```

```
webPort.send(queryBooksHerge);
```

```
alt {
    [] webPort.receive(hergeDBQueryResultsPage(booksTables)) {
        setverdict(pass)
    }
    [] webPort.receive {
        setverdict(fail)
    }
}
```

Template transformation function

at the abstract layer

```
function transformDBResultsIntoHTMLTables(ItemsType theDBItems)
                                          return TableSetType {
  …
  theTableRows[0] := { cells := {"author", "title", "price" }};
  for(i:=0; i < numOfDBRows; i:=i+1) {
    if(ischosen(theDBItems[i].booksTable)) {
      aBook := theDBItems[i].booksTable;
      aRow := {
        cells := { aBook.author, aBook.title, myFloat2str(aBook.price) }
      };
      theTableRows[i+1] := aRow;
    }
  }
  theTable := { rows := theTableRows };
  tables[0] := theTable;
  return tables
}
```

Template transformation remarks

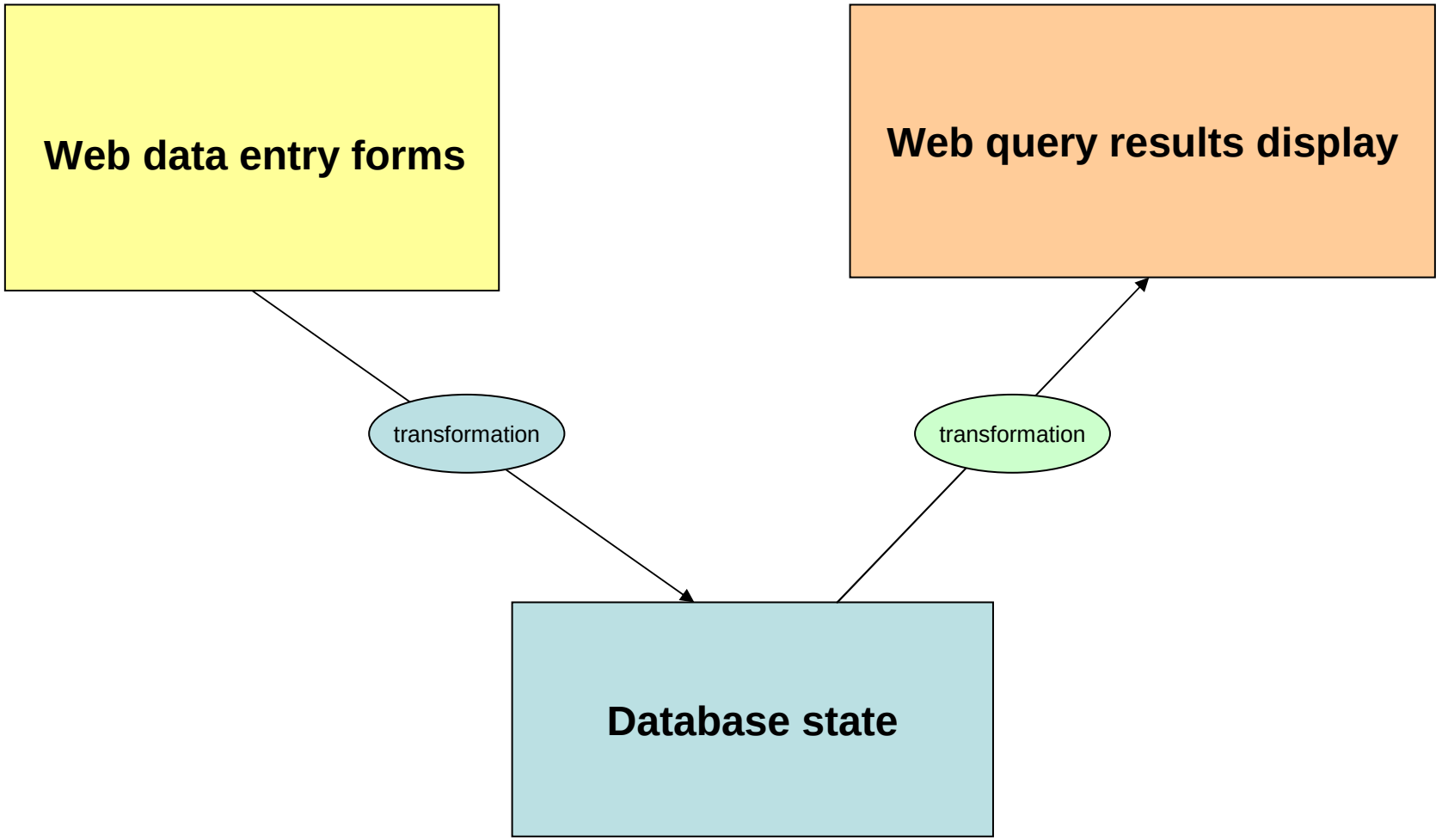
- The template transformations are obviously **had hoc**.
- It is application specific.
- It is not a framework.
- But the important thing is that they are all handled at the **abstract layer**
  - Thus using **only one language**, TTCN-3,
  - and **only one framework** for abstract typing, codecs and parametric templates.

Tests combinations

Double integration testing

- The two separate integration tests we have described so far can be combined.
  - Perform the web data entry test against the data base state.
  - Use the resulting database state for the web query test.

Two way transformations



Database application testing framework

- The combination of abstract data type strategy and concrete layer independence strategy constitutes a framework.
- The tester can re-use both abstract and concrete layer elements for an infinite variety of web/database application testing.

Framework details

- Abstract layer (TTCN-3)
  - The HTML typing
  - The SQL results set typing
- Concrete layer (Java & API)
  - The HTML codec
  - HTML test adapter
  - The SQL codec
  - SQL test adapter

Test development effort

- Takes place only at the abstract layer
- Consists in:
  - Crafting TTCN-3 templates
    - Analogous to filling a form
    - Carefully structuring templates to maximize re-use
  - Crafting template transformation functions
  - Crafting test behavior as reachability trees
- At all times, avoids distraction caused by concrete layer implementation problems.

Value of using TTCN-3

- The real value of using TTCN-3 is beyond mimicking unit testing
- It is found in the specification of a complex system that consists of various components that perform different services
  - data base services, web services, user interface services.
- The separation of concerns that TTCN-3 supports enables us to specify test suites strictly at the abstract level

Conclusions

- TTCN-3 is an efficient language for database application testing
- The codec can be relegated to a framework

Contact

- Bernard Stepien: bernard@site.uottawa.ca
- Liam Peyton: lpeyton@site.uottawa.ca