Content Management for Declarative Web Site Design

Richard Cooper and Michael Davidson

Computing Science

University of Glasgow

Talk Overview

- Motivation for uniform declarative approach to web site design
- A declarative component model
- The data source aspect
- An abstract model for data sources
- A hybrid data source implementation model

Motivation

- Web site design is usually achieved by
  - unstructured low level implementation
  - or the use of interface design tools creating unmaintainable implementations
  - or the use of Content Management Systems with complex structures
- We propose
  - a simple, systematic and comprehensive declarative model for specifying the site
  - familiar authoring tools
  - late decisions on the implementation technology, such as database or server side middleware
  - automatic creation of the implementation

Web Site Construction

Write Client Side Scripts

Javascript

Design site structure

Design Interaction

XHTML Forms + Middleware

Write Static Pages

XHTML

Layout Pages

SMIL?

Write Dynamic Pages

Middleware Product

SQL

CSS

Add Style

Build Content Management

JDBC

Build Data Source

Problems with this

- No integration
- Poor maintainability
- Re-use hard to do
- Lots of languages which change
- Design and implementation tightly bound
- However, separate development of components is good

Declarative Web Site Design

- The declarative approach promises the ability to describe a site at high level and to generate the details
  - Strudel separates the management of data from structure
  - Tiramisu separates design from implementation
  - Holland and Kumar – *Component-based web Page Composition*
    - hierarchy of presentation components
    - separate layout manager and renderers

An Integrated Approach

- Use one structure to describe all constituent parts
  - since all are software components
- Generate the implementation
  - ensuring that the components contain sufficient information
  - this permits late decisions on technology
- Use an O-O approach
  - to aid maintainability and re-use

Components

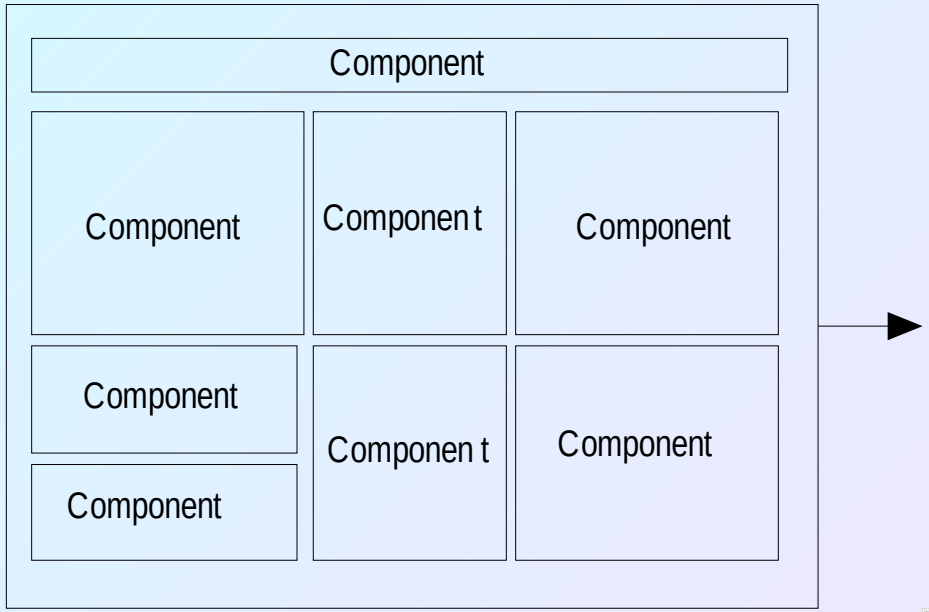
- A component is very simply described as an object having a number of parameters
  - abstract components have uninstantiated parameters
  - concrete components have values for all mandatory parameters
  - documentation parameters for all components – author, date of creation, description, etc.
  - representation in XML
- Generation proceeds from concrete components

Component Maintenance

- Component creation is by inheritance using two methods
  - instantiating parameters
  - adding new parameters

- Example
  - The title of a page can be fixed
    - making a more concrete component
  - The most general page component can be extended with parameters for an image, a heading and a series of text blocks
    - making a more detailed but still abstract component

Abstract and Concrete Components



Abstract component

Specialised component

Using the Model

- Each part of the site is designed from abstract components and each stage of the specialisation may be stored
- This is achieved using tools
  - permitting the separate specification of each aspect
  - using familiar techniques
- But the product of the tools is maintainable
  - being held in an integrated manner
  - and being declarative and documented

Examples

- A web page using a specific style sheet
- A table with particular columns
- A block including a specific place for an image, a heading and a piece of text
- i.e. any reusable structure, e.g.
- A data source with a fixed set of views but no particular data nor implementation

A Component Hierarchy

- At the top is *Component*
  - This has the parameters such as *name*, *parent*, *description*, *author*, *version*, *date* and *implementation*
  - implementation is probably needed an optional catch-all for placing code which is hard to generate

- Below this are abstract components for the main categories of site constituent
  - web site
  - layout (and region)
  - visible
  - script
  - web page
  - style sheets (and styles)
  - data source (and views)

The Main Page Constituents I

- Web Site
  - has title, style sheet, front page & a set of web pages
- Web Page
  - has a title, a style sheet, a set of meta-data and a sequence of visible components
  - dynamic pages also have a data source and a set of place holders for content
- Layout
  - c.f. SMIL – a set of regions
  - a Region is a portion of the visible page into which a visible component is placed

The Main Page Constituents II

- Style
  - a set of name, value pairs, i.e. comes from CSS
  - but can be used for anything (better called Map)
- Visible
  - any XHTML fragment appropriate for the body of a page (XHTML DTD used for the hierarchy under this)
- Script
  - abstract description of a script hopefully permitting frequent operations to be described (*not worked out*)

Data Sources

- A *Data Source* component describes the location of site content
- It is parameterised by a set of connection parameters and a set of views
  - each can be separately specified
- A *View* is a query and includes:
  - the returned metadata
  - whether it is single or multi-valued
  - whether it is a query or update
  - and the query string and an error message

Example

```
$db = mysql_connect(“NN", “DD", “PPPP") or die ("Could not connect");
$nameQuery = "SELECT Name FROM Advisor WHERE ID = '$Advisor'";
$nameResult = mysql_query($nameQuery,$db) or die (“Name query error");
$nameRow = mysql_fetch_array($nameResult);
echo “<h3>Advisees of $nameRow[‘Name’]</h3>”;

$query = "SELECT Name, Matric FROM Student
            WHERE Advisor = '$Advisor' ORDER BY Year, Name";

$result = mysql_query($query, $db) or die (“Student query error");
echo "<table>";
echo "<tr> <th>Name</th> <th>Matric</th> </tr>";

while ($row = mysql_fetch_array($result))
{
    echo "<tr><td>$row["Name"]</td>
            <td>$row["Matric"]</td></tr>";
}

echo "</table>";
```

Analysis

Implementation Detail

Data Source

```
$db = mysql_connect("UN", "DN", "PASS") or die ("Could not connect");
$nameQuery = " SELECT Name FROM Advisor WHERE ID = '$Advisor' ";
$nameResult = mysql_query($nameQuery,$db) or die ("Name query error");
$nameRow = mysql_fetch_array($nameResult);
echo "<h3>Advisees of $nameRow['Name']</h3>";

$studentQuery = "SELECT Name, Matric FROM Student
            WHERE Advisor = '$Advisor' ORDER BY Year, Name";

$result = mysql_query($studentQuery, $db) or die ("Student query error");
echo "<table>";
echo "<tr> <th>Name</th> <th>Matric</th> </tr>";

while ($row = mysql_fetch_array($result))
{
    echo "<tr><td>$row["Name"]</td>
            <td>$row["Matric"]</td></tr>";
}

echo "</table>";
```

Views

Visible Fragments

Data Source Component

- The abstract Data Source Component has the parameters:
  - Name – maps to DN on the previous slide
  - Owner – maps to UN on the previous slide
  - Password – maps to PASS on the previous slide
  - Views – a set of view components
  - Kind – relations, XML, ????
- There are abstract sub-types for each data source kind and each data management product

View Components

- A view component represents the results of a query which can be run on the data source

- Parameters
  - data source – *get the red stuff from the green stuff*
  - query – the string which is run against the data source
  - ColNames – the names of columns returned
  - ColTypes – the types of columns returned
  - QueryParams – the values of any parameters in the query
  - Card – does the query return one or many records
  - READorWRITE – querying or updating?

Typical Visible Components

- Visible components include abstract types for each major XHTML component type in <body>
  - blocks and inlines are the major abstract components

- In the example, we use:
  - SingleQueryResult – a subtype of *Inline* with the parameters *View* and *QueryParameters*
    - *get the green stuff from the blue stuff*
  - DynamicTable – a subtype of table with the parameters *View* and *QueryParameters*

The Need for an Abstract Data Model

- The example shows SQL access to an RDB
- However, the data source may be in another format
  - XML, OODB, Spreadsheet, etc.
- We therefore require a data representation which is not bound to any specific implementation structure
  - so that the queries can be expressed
  - and the content management
  - we use an entity based data model

The Abstract Data Model

- Data sources are described in terms of an abstract data model
  - a **schema** is a set of entity types
  - each **entity type** has a set of **properties**
    - either base type or entity type
    - single or multi-valued
    - unique, non-null or key
    - inverses can be specified
- The abstract model is implemented in terms of
  - relations or XML or **both**
    - an algorithm determines which to use
  - also is extensible – e.g. addition of a *Gender* type

Example

Entity	Property	Domain	Imp	pk		Null	Unq	Inverse
Book	ISBN	String(20)	R		1			
Book	Title	String (30)	R		1			
Book	PubDate	Date	R		1			
Book	Author	Author	R		M			Author.Works
Book	PublishdBy	Publisher	R		1			Publisher.Publishes
Author	ID	Integer	R		1			
Author	Name	String (20)	R		1			
Author	BirthDate	Date	R		1			
Author	DeathDate	Date	R		1			
Author	Gender	Gender	R		1			
Author	Photograph	Image	F		1			
Author	Works	Book	R		M			Book.Author
Publisher	ID	Integer	X		1			
Publisher	Name	String (20)	X		1			
Publisher	Address	String (50)	X		1			
Publisher	Publishes	Book	X		M			Book.Publisher

Mappings to Implementation Structures

- Relations
  - Standard ER techniques map an abstract schema to a set of **create table** statements
- XML
  - Kleiner & Lipack show how such a model maps to DTDs
- Hybrid model
  - RDB can identify XML objects using tag + ID
  - XML can identify RDB objects using table name and Pkey
  - The important issue is not to lose typing and to ensure that the generation software has enough to go on

Example

- Given Book and Publisher entity types related by an inverse relationship, you can implement by:

create table Book(ISBN Varchar2(20) **Primary Key**,
Title Varchar2(30) **Non Null**,
PublishedBy Number2 **references** Publisher.ID)

create table Publisher(ID: Number2 **Primary Key**,
Name Varchar2(20) **Non Null**,
Address Varchar2(50))

or

```
<!ELEMENT Book EMPTY>
        <!ATTLIST Book ISBN ID #REQUIRED>
        <!ATTLIST Book Title CDATA #REQUIRED>
        <!ATTLIST Book PublishedBy IDREF #IMPLIED>
<!ELEMENT Publisher EMPTY>
        <!ATTLIST Publisher ID ID #REQUIRED>
        <!ATTLIST Publisher Name CDATA #REQUIRED>
        <!ATTLIST Publisher Address CDATA #IMPLIED>
        <!ATTLIST Publisher Publishes IDREFS #IMPLIED>
```

Hybrid Representation

Conventional names

- If Book is in XML while Publisher is in an RDB, you can do the following:

  <!ATTLIST Book RDB$PublishedBy CDATA #IMPLIED>

and

create table Publisher(ID: Number2 **Primary Key**,
Name Varchar2(20) **Non Null**,
Address Varchar2(50)
XML$Publishes Varchar2(100))

- Sample data:

  attribute in BOOK element:
  RDB$PublishedBy = “Publisher:23”

  data value in XML$Publishes table
  “Book:1224, 3456, 5678”

Pkey

IDs

Automatic Implementation Selection

- We have also implemented in the schema editor and algorithm for automatically choosing which mix of RDBs and XML is most suitable

- Intuitions
  - Given few elements containing large blocks of unstructured text we want XML.
  - Given many elements with small data types we want a relational table
  - If we have complex constraints we want a relational table

The Alogorithm

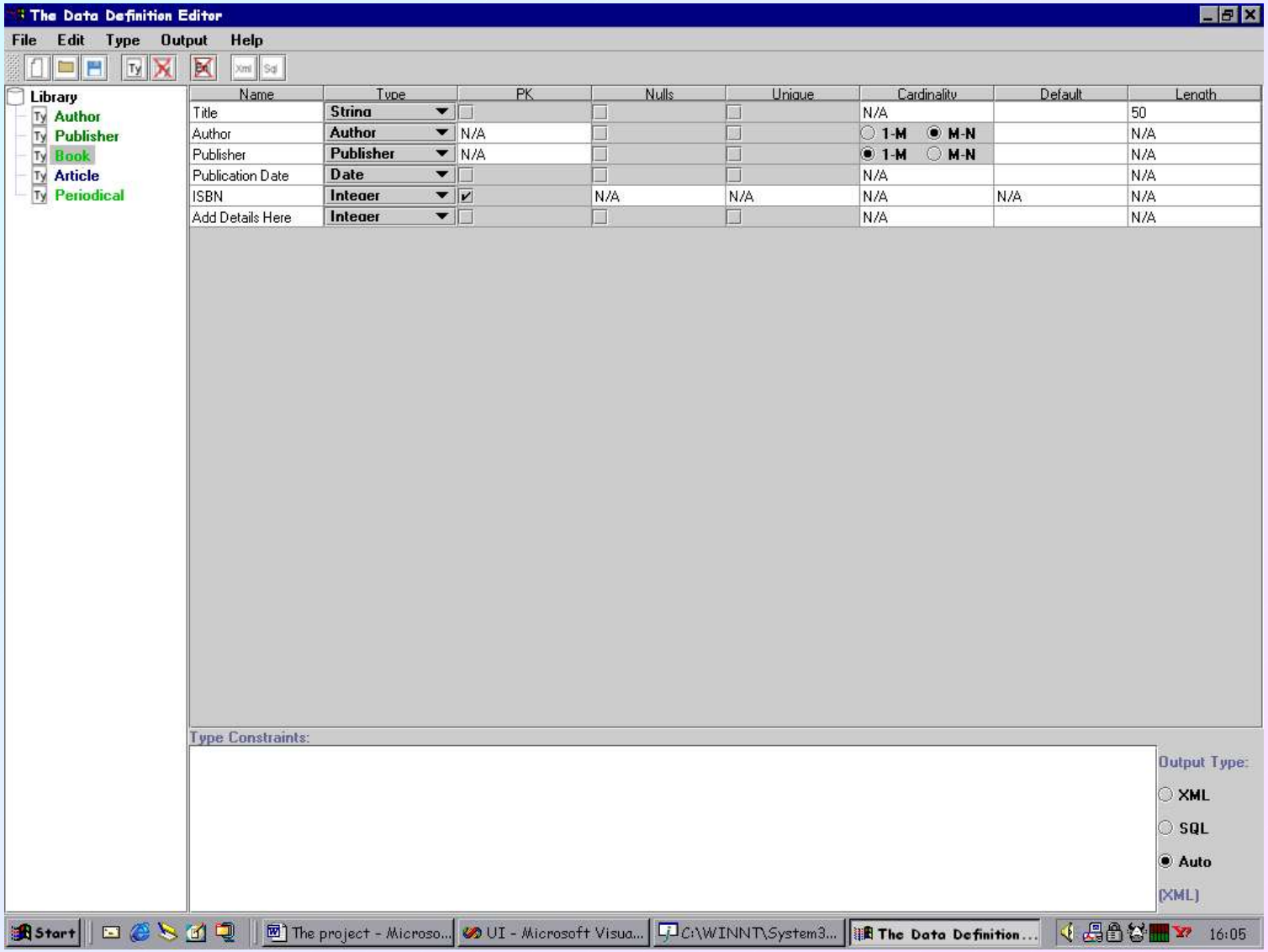
- For each entity type we add up the following weights:

SQL Property		**XML property**	
Each non text property in type	5	Each long text element in type	20
Each short text element in type	1	Each reference to another type (1-M)	1
Each simple constraint	1	All element types are text (N = number of types)	2N
Has check constraints	50		

Software Support

- Schema Editor (Michael Davidson)
  - manages the creation and maintenance of data source schemata
- Browser (ChengCheng Zhou)
  - permits the browsing of data howvere it is implemented
- Query Language (Si Ying Meng)
  - permits the expression of queries over the data

The Schema Editor




ORACLE: CV
ID
RESEARCHITEM
2
ResearchItem 2

Query Support

- The query language we have chosen is similar to OQL and has the basic form:

  SELECT $path_1$, $path_2$, …, $path_m$

  FROM $EntityTypeName_1$, $EntityTypeName_2$, …, $EntityTypeName_n$

  WHERE .....logical expression involving paths and constants

- This maps simply to:
  - SQL
  - XPATH

Example

```
SELECT  Works.Title, Works.PublishedBy.Name
    FROM  Author
            WHERE  Name = ‘Jane Austen’;
```

becomes:

```
SELECT  book1.Title, publisher1.Name
    FROM  Author author1, Writes writes1, Book book1, Publisher
                                                          publisher1
            WHERE  author1.Name = ‘Jane Austen’ and
                        and write1.Book = Book1.ISBN
                        and Book1.PublishedBy=publisher1.ID;
```

or

```
    Book[child::author=”Jane Austen”]::Title,
    Book[child::author=”Jane Austen”]::Publisher::Name
```

Much To Do

- Integrating other implementation models
- Integrating the three data management tools
- Switching to XML Schema
- The Generation software
- Tantalising thought
  - If the declarative model is published as part of the site with the views clearly available, does this fixed the hidden web problem