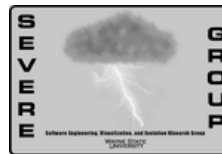


Using Traceability Links to Assess and Maintain the Quality of Software Documentation

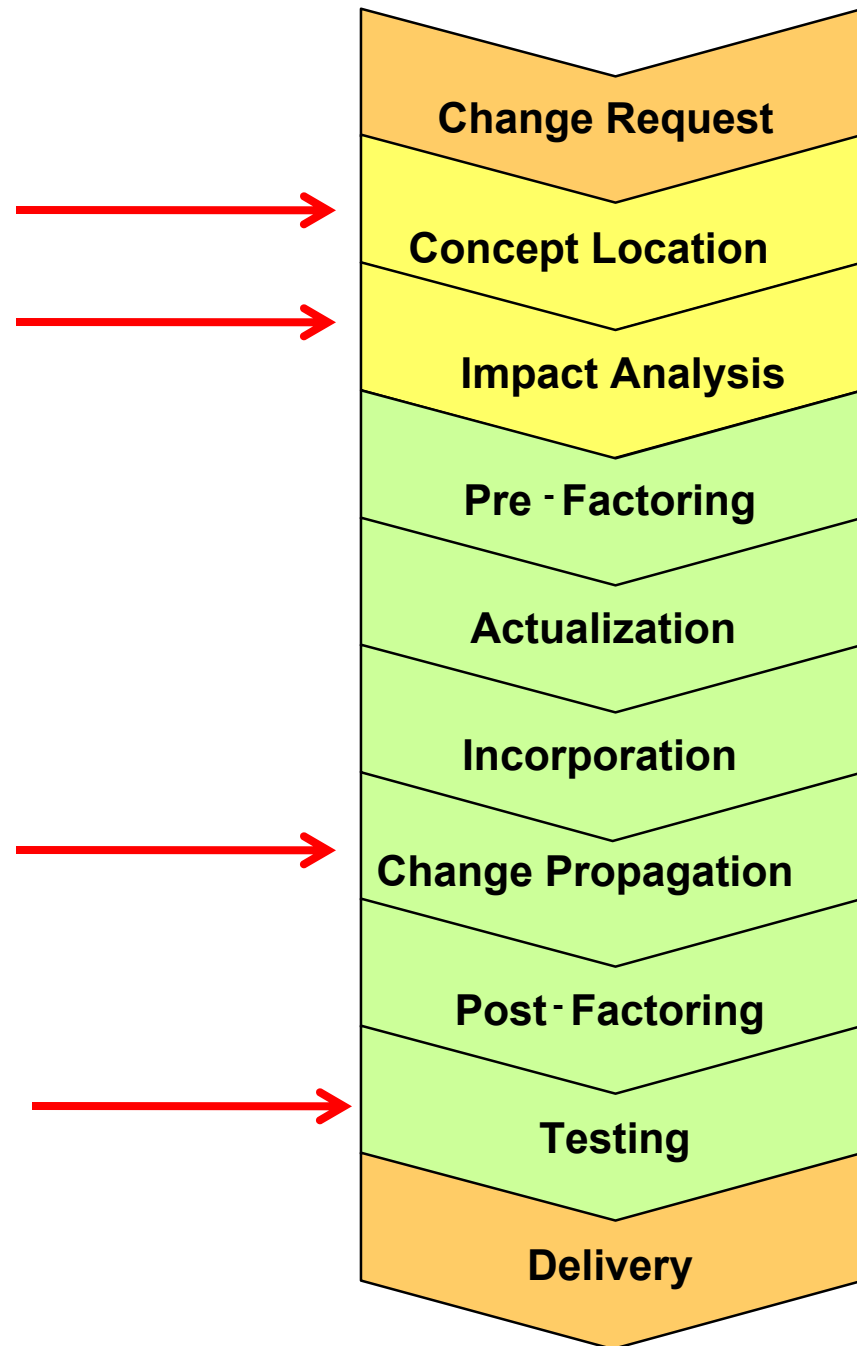
Denys Poshyvanyk and Andrian Marcus

International Symposium on Grand Challenges in Traceability
(GCT'07 / TEFSE'07)

Lexington, Kentucky



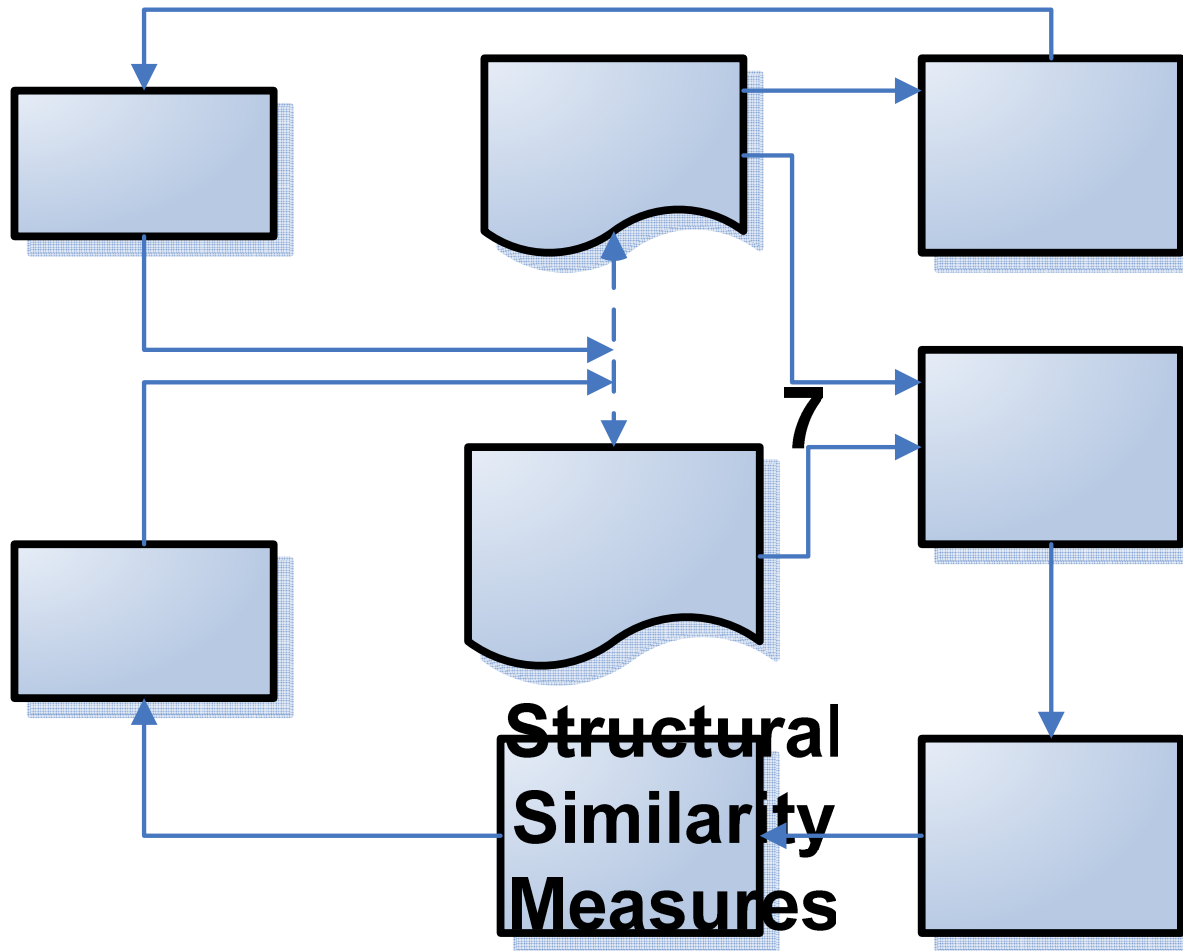
Incremental Change



Problem Description

- Documentation structure does not reflect in many cases the structure of the source code (missing links among related sections in the docs)
- As source code evolves (structure changes), the documentation should also reflect those changes within its internal structure (e.g. major refactoring of the source code)

Proposed Approach



Applications

- On systems with existing traceability links between source code and documentation
- To recover initial traceability links to enhance the structure of documentation

Improving Documentation in LEDA

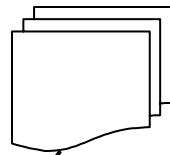
- LEDA implemented in C++
- 115 manual sections in English
- 219 classes
- Traced links from documentation to the source code

Coupling Measures

- Coupling measures computed with Columbus [Ferenc'04] :
 - CBO, RFC, MPC, DAC, ICP, ACAIC, OCAIC, ACMIC, OCMIC
- Conceptual coupling measures (CoCC) computed with our IRC³M tool

Supporting Link Recovery

User manual



```

// Base class for all nodes
class Node {
public:
    Node() {}
    Node(int val) : val(val) {}
    Node(int val, Node* next) : val(val), next(next) {}
    int val;
    Node* next;
};

// Class A implementation
class A : public Node {
public:
    A(int val) : Node(val) {}
};

// Class B implementation
class B : public Node {
public:
    B(int val) : Node(val) {}
};
    
```

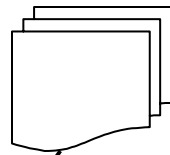
Class A

Class B

Class A
Class D
...
...
Class B

Supporting Link Recovery

User manual



```

// Base class for all nodes
class Node {
public:
    Node() {}
    Node(int val) : val(val) {}
    Node(int val, Node* left, Node* right) : val(val), left(left), right(right) {}
    virtual ~Node() {}
    int val;
    Node* left;
    Node* right;
};

// Binary tree implementation
class BinaryTree {
public:
    BinaryTree() {}
    BinaryTree(int val) : root(new Node(val)) {}
    BinaryTree(int val, Node* left, Node* right) : root(new Node(val, left, right)) {}
    virtual ~BinaryTree() {}
    Node* root;
};

// Insertion
void BinaryTree::insert(int val) {
    insert(root, val);
}

void BinaryTree::insert(Node* node, int val) {
    if (!node) return;
    if (val < node->val)
        insert(node->left, val);
    else if (val > node->val)
        insert(node->right, val);
    else
        node->val = val;
}

// Search
Node* BinaryTree::search(int val) {
    return search(root, val);
}

Node* BinaryTree::search(Node* node, int val) {
    if (!node) return nullptr;
    if (val == node->val)
        return node;
    if (val < node->val)
        return search(node->left, val);
    if (val > node->val)
        return search(node->right, val);
    return nullptr;
}

// In-order traversal
void BinaryTree::inorder() {
    inorder(root);
}

void BinaryTree::inorder(Node* node) {
    if (!node) return;
    inorder(node->left);
    cout << node->val << " ";
    inorder(node->right);
}

// Pre-order traversal
void BinaryTree::preorder() {
    preorder(root);
}

void BinaryTree::preorder(Node* node) {
    if (!node) return;
    cout << node->val << " ";
    preorder(node->left);
    preorder(node->right);
}

// Post-order traversal
void BinaryTree::postorder() {
    postorder(root);
}

void BinaryTree::postorder(Node* node) {
    if (!node) return;
    postorder(node->left);
    postorder(node->right);
    cout << node->val << " ";
}

// Level-order traversal
void BinaryTree::levelorder() {
    levelorder(root);
}

void BinaryTree::levelorder(Node* node) {
    if (!node) return;
    queue<Node*> q;
    q.push(node);
    while (!q.empty()) {
        Node* n = q.front();
        cout << n->val << " ";
        if (n->left) q.push(n->left);
        if (n->right) q.push(n->right);
        q.pop();
    }
}

// Main
int main() {
    BinaryTree tree;
    tree.insert(10);
    tree.insert(5);
    tree.insert(20);
    tree.insert(3);
    tree.insert(7);
    tree.insert(15);
    tree.insert(12);
    tree.inorder();
    tree.preorder();
    tree.postorder();
    tree.levelorder();
    return 0;
}

```

Class A

Class B

Class A
Class B
...
...
Class D

Other Applications

java.util

Class `ArrayList<E>`

```
java.lang.Object
├── java.util.AbstractCollection<E>
│   └── java.util.AbstractList<E>
│       └── java.util.ArrayList<E>
```

```
public class ArrayList<E>
    extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

See Also:

[Collection](#), [List](#), [LinkedList](#), [Vector](#), [Serialized Form](#)

Future Work

- Other types of artifacts, e.g. design docs, requirements
- Case studies on single version of software
 - to evaluate the impact of using structural/conceptual similarities to refine the structure of documentation
 - which combination of coupling measure can adequately reflect the structure?
- Case studies on multiple versions of software
 - to evaluate the impact of structural similarities in source code due to incremental changes