Testing Business Component Systems

T. Toroi, A. Eerola, J. Mykkänen

Report A/2002/1

ISBN 951-781-261-2
ISSN 0787-6416

UNIVERSITY OF KUOPIO
Department of Computer Science
and Applied Mathematics

P.O.Box 1627, FIN-70211 Kuopio, FINLAND

Testing business component systems

Tanja Toroi
University of Kuopio
Department of computer science and applied mathematics
P.O.B 1627, FIN-70211 Kuopio
+358-17-163767
tanja.toroi@cs.uku.fi

Anne Eerola
University of Kuopio
Department of computer science and applied mathematics
P.O.B 1627, FIN-70211 Kuopio
+358-17-162569
anne.eerola@cs.uku.fi

Juha Mykkänen
University of Kuopio
Computing Centre
HIS Research & Development Unit
P.O.B 1627, FIN-70211 Kuopio
+358-17-162824
juha.mykkanen@uku.fi

ABSTRACT
Nowadays it is demanded that software system fulfils more quality requirements, especially in the health care and other safety critical systems. In this paper we present an effective and practical method for testing business component systems step by step. We utilize components of different granularity levels. The advantages of component-based systems are the possibility to master development and deployment complexity, to decrease time to market, and to support scalability of software systems. Also the great number of dependencies which occur in object-oriented approach can be mastered better, because majority of the dependencies between classes remain inside one component where the number of classes is much less than in the total system or subsystem. The abstraction levels decrease the work needed in testing, because the testing work can be divided into sufficiently small concerns and the previously tested components can be considered as black boxes, whose test results are available. Furthermore, errors can be easily detected, because not so many components are considered at one time.

In our method, components of different granularities are tested level by level. The idea of the method is that at each level white box testing and black box testing occur alternately. We define test cases based on component granularities at distributed component, business component and business component system level. Test cases are derived from use cases or contracts. We use a dependency graph, which shows dependencies between the same granularity components. The dependency graph is used to assure that the whole functionality of the component has been covered by test cases.

Categories and Subject Descriptors
D.2.5 **[Software Engineering]**: Testing and Debugging – *Testing tools.*

D.2.9 **[Software Engineering]**: Management – *Software quality assurance.*

General Terms
Design, Reliability, Theory, Verification.

Keywords
Testing, distributed component, business component, business component system, distribution tiers, layers, interfaces, dependency

1. INTRODUCTION
Component software technologies have been exploited in software industry more and more. The crucial objectives in this approach are decreased time to market, efficiency in software production, quality and reusability of software and its components. Distribution of software and its production is the fact today. Software providers do not necessarily want to implement all the properties of the system by themselves but they want to specialize in their strategic competitive edge and buy other properties as ready-made COTS (commercial-off-the-shelf) components. In situations like this testing and documentation are even more important than in conventional software projects. Customers of software require quality such as correctness and reliability in addition to functional properties of the software system. This is especially true in health care information systems and in other safety critical systems.

There are several definitions of the concept of testing [19]. In this research, testing means that the software is executed in order to find errors. Thus debugging process is not included in testing, although it is a quite near to it. In order to develop quality business components, system testing must verify that the software product operates as documented, interfaces correctly with other systems, performs as required, and satisfies the user's needs [26].

Although component technologies have been widely introduced in research and industry only a few investigations have been dedicated to testing of component based systems. Even less have integration testing and interoperability testing been considered in research papers. Unlike traditional software developers, component developers are in most cases independent software vendors who may not know all the future uses of their components [26]. Only the known uses can be tested beforehand. In order to increase the reusability we must carefully define the provided interfaces and constraints, i.e. runtime environment and dependencies, of the components.

Traditional techniques such as data flow testing, mutation testing, and control flow testing are not very well applicable when testing component systems. Testing distributed component-based systems requires that heterogeneity, source code availability, maintainability and complexity need to be considered [27].

Information systems must collaborate with each other in order to fulfil the requirements of stakeholders. The collaboration is achieved by integrating the systems. The integration at the system level can be done utilizing different interaction models, for example, integrated, bus-based, bridged, and coordinated [5].

Users require high quality and reliability. As a consequence, the information system needs a comprehensive inspection and testing process. Testing the interoperability and quality of an information system as a big bang in a deployment or introduction stage is difficult, costly or even impossible. The reasons of this are the heterogeneity, complexity, diversity of systems, and reuse of COTS, for which code is not available. For producing quality software the quality must be emphasized right at the beginning of the projects and the quality assurance must occur similarly while building and buying the components of the software system. For this reason, the software process should include high quality inspection and testing policies, which verify that all the functional and non-functional requirements of the stakeholders will be in the final product and the product does not have not-needed properties. Thus forward and backward traceability is required [4]. The requirements of stakeholders are gathered and then used, at the analysis level, in order to derive use cases and conceptual class hierarchy, which are then used as a starting point at the design phase [18,2].

Wu et al introduced that errors in component-based system can be inter-component faults, interoperability faults in system level, programming level, and specification level, and traditional faults [28]. They have also presented a maintenance technique for component-based software, where static analysis is used to identify the interfaces, events and dependence relationships that would be affected by the modifications. The results obtained from the static analysis are used to help test case selection in the maintenance phase. In this technique, they do not separate different dependency relationships.

We propose an improvement where dependencies inside one component do not interfere with external dependencies and present an effective and practical method for testing business component systems step by step. We consider functional requirements only. We utilize components from different granularity levels defined in Herzum and Sims [5]. The granularity hierarchy means that a business component system is a composition of business components, which in turn are compositions of lower level components. The advantages of this approach are the possibility to master development and deployment complexity, to decrease time to market, and to increase scalability of software systems. Also the great number of dependencies which occur in object-oriented approach [25] can be mastered better, because majority of the dependencies between classes remain inside one component where the number of classes is much less than in the total system or subsystem. The abstraction levels decrease the work needed in testing, because the testing work can be divided into sufficiently small concerns and the previously tested components can be considered as black boxes, whose test results are available. Furthermore, errors can be easily detected, because not so many components are considered at one time.

The remainder of the paper is organized as follows: In Section 2 we describe the component properties and the component granularity levels. Our testing method in general is given in Section 3 and in Section 4 is the testing process for different granularity components. Related work is considered in Section 5. A conclusion is finally given in Section 6.

2. PROPERTIES OF COMPONENTS

2.1 Interfaces and Contracts

The definition of a component stresses its autonomy [5, 22]. Each component forms a cohesive and encapsulated set of operations (or services). A component should have minimal and explicitly defined relationships with other components minimizing coupling [13]. The semantics of the relationships should be defined as well as the syntax. This leads to the definition of interfaces of the component, which hide internal properties of the component. The *interface* of the component specifies the list of provided operations, for each operation the list of parameters, and for each parameter the type and direction (in or out). Interface Definition Language (IDL) [14] is usually utilized in the interface specification, where semantics can be described using preconditions, postconditions, and invariants [22].

However, components can not be isolated. A component should not be too wide and complicated. Thus it can not perform all the things by itself, but it collaborates with other components: First, a component may call other components by sending a message. In this paper, we consider only synchronous messaging. Thus the caller of the operation waits for the result. Each interface of the component has a dependency relationships of its own. Second, a component needs an execution environment, i.e. a socket, into which it plugs and within which it executes. The provided interfaces, the required dependencies for each interface and the specified execution environments of the component form a *contract* between a called component and a component caller [22]:

Contract = interface, dependencies, environment, specification.

A component can have several contracts. Different customers want different properties and at the maintenance stage version management can be solved by making new versions of the contracts. Similarly contracts are a profitable document between the producer of a component and customers, who want to buy the component. This contract can be utilized in build-time, too.

2.2 Granularity of Components

By partitioning a given problem space into a component form we utilize, in this research, the business component approach introduced by Herzum and Sims [5]. Components of different granularities, i.e. distributed component, business component, and business component system are described in the following chapters.

2.2.1 Distributed Component

The *lowest granularity* of software component is called a distributed component. A *distributed component* (DC) has a well-defined build-time and run-time interface, which may be network addressable. A DC can be deployed as a pluggable binary component to the socket given in the contract. Further a distributed component may have dependency relationships to other components. A DC can have one or more interfaces, which define the operations component offers and the parameters needed when calling the component. Thus an interface of DC can be defined as follows:

Interface = (operation, (parameter, type, [in|out])*)*

User interface implementation should be separated from business logic implementation and database access. This leads to the definition of categories for distributed components, i.e. user DC, workspace DC, enterprise DC and resource DC. User DC differs from the other DCs because it does not have network addressable interface but it has user interface [20]. In object-oriented approach the distributed component usually consists of classes and relationships between them, but traditional programming approaches can be used, too. Thus a DC hides the implementation details from the component user. Component technologies offer usually a degree of location transparency and platform and programming language neutrality. For example, a distributed component could be lab order entry (see Fig. 1).

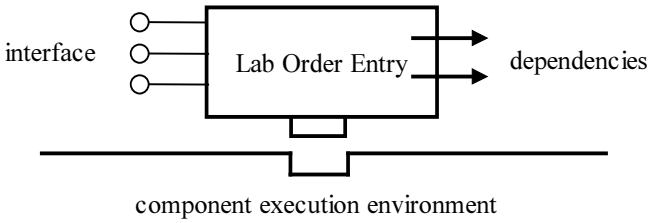


Figure 1. Lab Order Entry distributed component

2.2.2 Business Component

The *medium granularity* of software component is called a business component. A *business component* (BC) consists of all the artifacts necessary to represent, implement, and deploy a given business concept or business process as an autonomous, reusable element of a larger distributed information system. From the functional designer's point of view a business component can consist of user, workspace, enterprise and resource tiers. Each tier consists of zero or more DC whose category is the same as the tier. User and workspace tiers form a single-user domain. Enterprise and resource tiers form a multi-user domain.

In figure 2 a business component and the execution environment (CEE) for that component is presented. A distributed component can send a message to a component, which is at the same or at the lower tier as itself. Cyclic messaging should be avoided, because it violates normalization. Events, for example, error messages, such as database access violation, can go from lower tier to the upper tier. From the above follows that a business component is a composition component, whose parts are distributed components. The runtime interface of a BC is the union of all interfaces of its

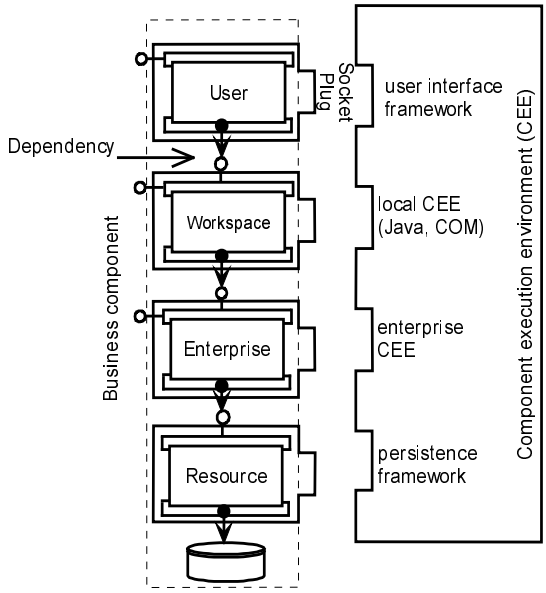


Figure 2. The business component and the component execution environment (Adapted from [5]).

distributed components, which are visible outside the boundaries of the BC. A business component could be, for example, lab test, which could contain different tests and the structure of results.

2.2.3 Business Component System

The *largest granularity* of software components is called a business component system. A *business component system* (BCS) is a composition component, whose parts are business components that constitute a viable system. The runtime interface of BCS is the union of all interfaces of its business components, which are visible outside the boundaries of the BCS. Business components can be classified into functional layers, for example, process, entity, and utility. In figure 3 there is Lab test business component system. At the process layer there is one business component, Lab test workflow. At the entity layer there are Lab test, Test result analyzer, Department and Patient business components. Utility layer business components include Lab test codebook and Addressbook. Database integrity manager and Performance monitor are auxiliary business components. As before, messages can go from the upper level to the lower level. The granularity of components gives controllability to the testing process as can be seen in the following chapters.

3. TESTING PROCESS

3.1 Testing Method

In this paper we present a testing method, where components are tested from the lowest granularity to the highest granularity. Testing process is analogous at each level and the test results of lower levels are available while testing the upper levels. Thus the test manager can see the quality of components while planning the test of a composition component. It is a known fact in practice that software parts that contain most errors are risky areas for

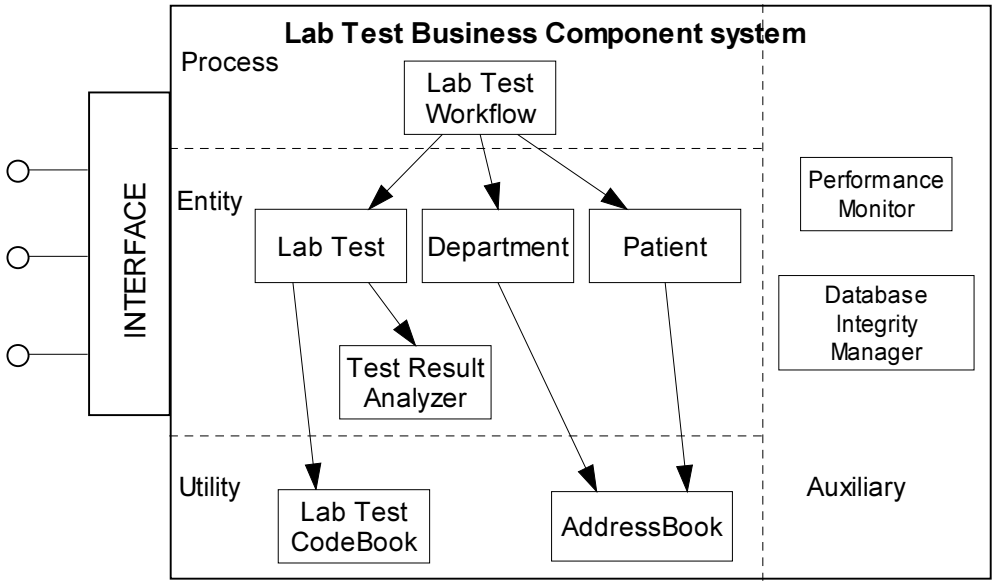


Figure 3. Lab Test Business Component System

errors also in the future. The idea of the method is that at each level white box testing and black box testing occur alternately:

- First, in unit testing phase the internal logic of the component is tested as a white box. Second, the external interface of the component is tested. Here the component is considered as a black box and it is tested in the execution environment, where it should work.
- In integration testing phase the component is considered as a composition component of its internal components. Now, the results of previous steps are available. First, the co-operation of internal components of the composition component is tested. Here, the internal components are black boxes. Second, the interface of the composition component is tested.

The alternation, presented above, is also true if we consider the role of component provider and component integrator:

The provider of the component needs white box testing for making sure that the component fulfils the properties defined in the contract and black box testing for making sure that the interface can be called in environments specified in contract.

The third-party integrator integrates self-made and bought components into a component-based system. He uses black box testing for making sure that he gets a product that fulfills requirements. Integration testing of self-made and bought components means that calling dependencies of components are considered. Thus from the systems point of view the internal logic of the system is considered. We denote this as white box testing although code of internal components is not available. At last external interfaces of the system are tested and here the total system is considered as a black box.

We can also consider a customer who buys a component-based system. He does not know the internal logic of the system. Thus he uses black box testing techniques for acceptance testing.

In the following chapters we first define test cases (chapter 3.2). When executing the system with carefully chosen test cases tester can see if the user requirements are fulfilled. Test cases are derived from the use cases or contracts. Furthermore, we need a dependency graph (chapter 3.3), which shows if all the parts of the system have been covered by the test cases.

3.2 Test Cases

3.2.1 Definition of Test Cases

A test case is generally defined as input and output for the system under test. Kaner [7] describes that a test case has a reasonable probability of catching an error. It is not redundant. It's the best of its breed and it is neither too simple nor too complex. A test case is defined in Rational Unified Process [9] as a set of test inputs, execution conditions, and expected results developed for particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. We insert the granularity aspect in the definitions of test cases:

- Test case at the business component system level is derived based on an action flow between business components and users.
- Test case at the business component level is derived based on a sequence of operations between distributed components.
- Test case at the distributed component level is derived based on a method sequence between object classes.

3.2.2 Use Cases and Contracts

Use cases and scenarios provide means for communication between users and a software system [6, 21]. Thus they are useful while deriving the responsibilities of BCS. The responsibilities of total BCS, which is considered as a composition component, are divided into responsibilities of each internal BC of BCS, recursively if needed. Thus we can suppose that for BCS and BC

we have use case descriptions, which show those needs of the stakeholders that are addressed to that component. With use cases we know how to actually use the system under test. Every use case must have at least one test case for the most important and critical scenarios.

However, use case diagrams and scenarios usually show only the communication between users and a software system. Thus the co-operation of human actors is not presented. We propose that the definition of use case diagram is extended to contain human actions as well as automated actions [8]. Consequently, action flows can be derived from these use case diagrams and it is possible to test that human workflow fits together with actions of BCS.

Next, we present examples of an action flow, an operation flow and a method sequence. They are used when test cases are derived for BCS, BC and DC. Examples are simplified, but they describe how abstraction levels differ when moving from BCS level to DC level. At the DC level test cases are the most accurate and detailed.

For example, an action flow of Lab Test BCS (see Fig. 3) could be following (actors in parenthesis):

- Create or choose a patient; (human and Patient BC)
- Examine patient; (human and Patient BC)
- Create a lab order; (human and Lab Test BC)
- Send the lab order to the lab; (human and Lab Test BC)
- Reception; (human and Lab Test BC)
- Take a sample; (human)
- Analyze the sample and return results; (human and Test Result Analyzer BC)
- Derive reference values; (Test Result Analyzer BC)
- Save lab test results and reference values; (human and Lab Test BC)

An operation sequence of Lab Test BC could be following:

- Input a patient number; (human and user DC)
- Find lab test results with reference values; (enterprise and resource DC)
- Output lab test results with reference values; (user DC)
- Evaluate results; (human)
- Decide further actions; (human)
- Send lab test results and advice for further actions to the department; (human, user and enterprise DC)

A method sequence of Lab Test user DC could be following:

- Input a patient number; (human and Patient class)
- Choose an activity lab test consulting; (human and Menu class)
- Send message “find lab test results” to enterprise DC; (proxy)
- Receive lab test results; (proxy)
- Output lab test results; (Lab Test class)
- Output reference values; (Reference class)

The other possibility to define test cases is to utilize contracts, specifying the operations offered by the components. In testing we must have a requirement document in which each operation of the interfaces has been described in detail. There we get input parameters of the operations and their domains. For each operation each parameter's input domain is divided in so called equivalence classes [15]. Equivalence partitioning is always the tester's subjective view and thus may be imperfect. If the operation has many input parameters then the equivalence classes for the whole input can be designed as a combination of the equivalence classes. This leads to an explosion in the number of equivalence classes. Test cases are selected so that at least one test input is selected from every equivalence class. So we get testing which is on one hand effective and covers customers' requirements and on the other hand it is not too arduous and complex. Redundancy is also as minimal as possible. For distributed components in resource, enterprise and workspace tier contracts may be the only possibility to define test cases. But it should be remembered that contracts do not specify the cooperation between more than two components. Thus they are not sufficient while testing action flow of stakeholders and the usability of the whole system.

Finally we check that test cases traverse all the paths of all the dependency graphs as presented in the next chapter.

3.3 Dependency Graph

3.3.1 General

Test cases, which are defined by use cases or contracts do not necessarily cover the whole functionality of the system. Besides use cases and contracts, we also need the dependency graph, which shows dependencies between the components of the same granularity level. If we do not know all the dependencies we do not know how to test them. We use the dependency graph to assure that the whole functionality of the component has been covered by test cases. Without dependency graph there may remain some critical paths that have not been executed or there may be some paths that have been tested many times. Redundant testing increases always the testing costs. Then test cases have to be removed. If there are paths, which are not traversed at all we must examine carefully if the components on those paths are needless for some reason or we have to add test cases so that all the paths will be traversed.

The following chapter describes how to compose the dependency graph and provides an example. The algorithm creates a dependency graph for each external interface of the composition component. So testing and maintaining component-based systems is easier than if we had only one big graph.

3.3.2 Dependency Graph Creation

We create a graph based on dependencies between different components in a composition component. Term composition component can either be a business component or a business component system. If it is a business component, dependencies are between distributed components and if it is a business component system, dependencies are between business components. A node in a graph represents a component. A directed edge from component A to component B means that

component A depends on component B (component B offers functionality to component A). The inner for-loop checks dependencies only at the same tier or tiers below because messages from one component to another go from upper to lower tier. The outer for-loop checks all the external interfaces the composition component has and creates a graph for each interface. Our algorithm follows the breadth first search algorithm.

Algorithm: Creates composition component dependency graphs.

Input: A composition component and its contracts.

Output: Composition component dependency graphs.

Variables:

- A set *called* includes all the components that have been traversed.
- A set *not_visited* includes components that have not been yet traversed.
- A set *targets* includes components that have already been as target components.

for each external interface of a composition component {

start with the component c_0 to which the interface refers;

create an edge from the interface to c_0;

not_visited = {all components in the composition component};

called = {c_0};

targets = ∅;

while not_visited ≠ ∅ and called - targets ≠ ∅ {

select *target* component from the (called - targets) set;

for each component c in the composition component in the same tier or below than *target* {

if *target* depends on a component *c* **then**

called = called ∪ *c*;

create an edge from *target* to *c*;

}

not_visited = not_visited - *target*;

targets = targets ∪ *target*;

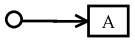
}

}

If BCS includes components, which can not be reached by any interface, those components do not belong to the graph. If there are cycles in the graph the algorithm reveals them as can be seen in the following example.

3.3.3 Example

In this chapter there is an example of using the previous algorithm. In figure 4 there are distributed components A - F and their dependencies in a 3-tiered business component. Let's start with the user level external interface. It refers to the component A.



// create an edge from the interface to c_0

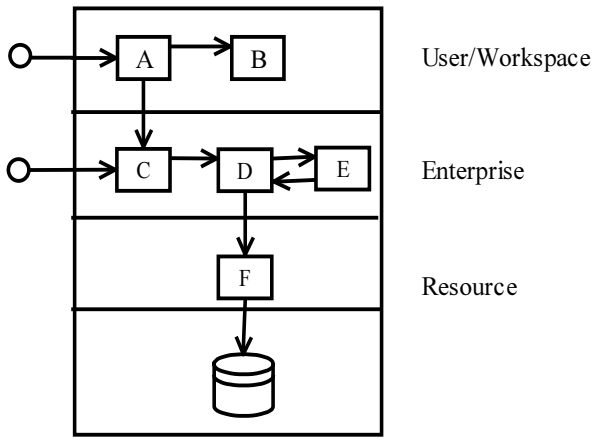
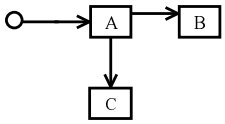


Figure 4. Graphical view of distributed components in a business component

not_visited = {A,B,C,D,E,F}

called = {A}
targets = ∅
not_visited ≠ ∅ and called - targets = {A}-∅ = {A}
target = {A}
called = {A} ∪ B = {A,B}

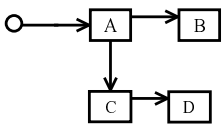
called = {A,B} ∪ {C} = {A,B,C}



component A has no other dependencies
not_visited = {A,B,C,D,E,F} - {A} = {B,C,D,E,F}
targets = ∅ ∪ {A} = {A}

not_visited ≠ ∅ and called - targets = {A,B,C}- {A} = {B,C}
target = B
no dependencies
not_visited = {B,C,D,E,F} - {B} = {C,D,E,F}
targets = {A} ∪ B = {A,B}

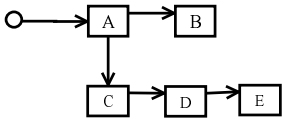
not_visited ≠ ∅ and called - targets = {A,B,C}- {A,B} = {C}
target = C
called = {A,B,C} ∪ D = {A,B,C,D}



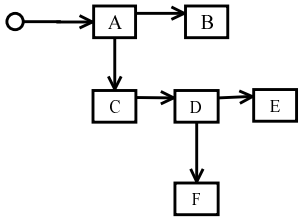
no other dependencies

not_visited = {C,D,E,F} - {C} = {D,E,F}
targets = {A,B} ∪ C = {A,B,C}

not_visited ≠ ∅ and called - targets = {A,B,C,D}- {A,B,C} = {D}
target = {D}
called = {A,B,C,D} ∪ E = {A,B,C,D,E}

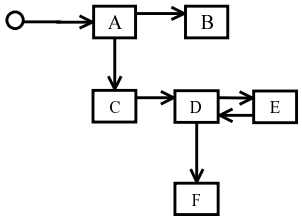


called = {A,B,C,D,E} ∪ F = {A,B,C,D,E,F}



no other dependencies
not_visited = {D,E,F} - {D} = {E,F}
targets = {A,B,C} ∪ D = {A,B,C,D}

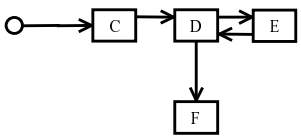
not_visited ≠ ∅ and called - targets = {A,B,C,D,E,F}- {A,B,C,D} = {E,F}
target = {E}
called = {A,B,C,D,E,F} ∪ D = {A,B,C,D,E,F}



no other dependencies
not_visited = {E,F} - {E} = {F}
targets = {A,B,C,D} ∪ E = {A,B,C,D,E}

not_visited ≠ ∅ and called - targets = {A,B,C,D,E,F}- {A,B,C,D,E} = {F}
target = {F}
no dependencies

Next the outer for-loop examines other external interfaces; in this case enterprise level interface, which refers to component C. Algorithm creates the following graph for this interface.



3.3.4 Selecting Test Cases

When we have created dependency graphs we have to create test cases based on those graphs. Test cases are created so that as many paths in a graph as possible are covered by one test case. This is called path coverage. Test suite satisfies 100% path coverage if all the paths in the dependency graphs have been executed. We should remember that the graphs are not very large because of components' granularity and because interfaces act as centralized connection points. So the complexity is lower if compared to the graphs of traditional or object-oriented software. We will study automatic test case selection in the future research.

4. TESTING COMPONENTS OF DIFFERENT GRANULARITIES

4.1 Distributed Components

Technical heterogeneity means use of different component technologies, programming languages and operating system environments. Productivity and flexibility of software implementation is increased by separating the functional logic from the runtime environment (socket) and the dependencies of the component technology, i.e. interface implementation and proxies which implement dependencies [5]. This profits the testing process, too. The functional logic is tested separately from interface implementation and proxies, which are substituted with driver and stub correspondingly if needed. Testing a distributed component depends on implementation. If a DC has been implemented by traditional techniques we can use traditional testing techniques and if it has been implemented by object oriented techniques we can use object oriented testing methods. In object oriented approach the functional code is usually implemented with classes and relationships between them. Testing means that methods and attributes of each class must be tested as well as the inheritance relationship between classes and association and aggregation relationships between objects. At the DC level test cases are usually derived from contracts. The execution of an operation, defined in the contract of DC, causes usually collaboration between several objects, which communicate with each other by sending messages. Thus the method sequence is one important subject to be tested. The objects dependency graphs can be derived analogously to the method presented in chapter 3.3 and it should be consistent with UML's collaboration diagrams defined in analysis stage. Furthermore, when we are testing a DC implemented by object oriented methods we can use several testing techniques [11]. For object state testing there are methods like [12, 23] and for testing inheritance relationship between classes there are methods like [16, 10].

Interfaces of DC must be tested locally and from the network. For user DCs the usability testing is important as well as the functionality tests. Testing resource tier DCs is more difficult if several databases or independent islands of data [5] are used.

4.2 Business Components

Vitharana & Jain [26] have presented a component assembly and testing strategy:

"Select a set of components that has maximum number of calls to other components within the set and minimum number of calls to other components outside the current set.

Of the remaining components, select a component (or a set of components) that has the maximum number of interactions with methods in the current application subsystem."

The strategy has been illustrated by an example but it has not been proved. However, the authors critique the strategy: The logical relationships between components should be taken into consideration while developing an integration test strategy.

We propose that in assembly and testing the business logic should be taken into account. Business components form a coherent set of properties, thus to test them as a whole is worthwhile. A business component is integrated from distributed components. Thus testing business component means:

- First, the integration testing of those distributed components, which belong to the business component is performed.
- Second, the external interface of the business component is tested.

While integrating a BC we propose that the integration strategy by Vitharana and Jain is modified as follows:

The assembly and testing go in two parallel parts:

In single-user domain part, the user and workspace tiers are integrated:

- It is profitable to start integration from user tier. Consequently, the comments from stakeholders are received as soon as possible.
- The workspace tier should be integrated next, because it is connected with the user tier.

In multi-user domain part, the resource and enterprise tiers are integrated:

- The resource tier is integrated first, because the DC in resource tier does not send messages to any lower tier.
- The enterprise tier is integrated next because it sends most messages to the resource tier.

Finally the total BC is integrated by combining the results of the above two parts. The above approach has several advantages. For example time to market decreases and controllability increases.

Testing business components is divided into two phases:

Phase 1:

BC's internal logic is considered using dependency graph similarly as before. Here each DC, which belongs to the business component is a black box, but BC itself is considered as a white box. Interfaces and dependencies are tested. The dependency graph is generated using the algorithm presented in chapter 3.3**.** If some of the DCs is not ready and has not passed through unit testing it is substituted with a stub.

The best way to derive test cases for BC is to utilize use cases. Because BC is a coherent set of operations of the business concept it is plausible that the most important and critical use cases of BC are defined at the analysis stage. Thus test cases can be built according to these use cases. The distributed components in user tier are the only components for which the user gives the input. For other BC-external interfaces the inputs come from some other systems or from the network. The values needed in BC´s internal dependencies are calculated inside DCs. Normal cases are tested before exceptions [21]. While considering exceptions the events go from the lower level to the upper level. For example, when an exception is noticed, a resource DC sends an event to an enterprise DC, which further sends an event to a user DC. This means that the algorithm forming dependency graph needs to be slightly modified while testing exceptions.

If use cases are not available, the contracts of the distributed components, which are visible outside the boundaries of BC are used. In this case the designer should decide the order of the operations.

Phase 2:

BC's external interface, especially the network addressability, should be tested. Here the BC is a black box. Partly the same test cases as before can be used. Now the internal logic is not considered, but the correspondence of operation calls with input parameter values is compared to the return values. Thus all the contracts of BC must be tested.

4.3 Business Component Systems

Business component system is assembled and tested using strategy presented in [26]. Utilities are often ready-made COTS, which need only acceptance testing. This means that the interface is tested and the component is seen as a black box. Entity BCs call utilities, thus entities are tested next. Process BCs call entities, thus they are tested last. Thus the order is: first utility BC, second entity BC, third process BC.

Testing business component system means:

- First, integration of those business components, which belong to the business component system is tested.
- Second, the external interface of business component system is tested.

In integration testing BCS's internal logic is considered. Here each BC of BCS is a black box, which has been unit tested. BCS itself is considered as a white box. Interfaces and dependencies are tested utilizing dependency graph similarly as before.

Test cases of BCS are constructed using use cases, which show the action flow of users of the BCS. Of course, it is possible that the inputs come from some other system, but these are considered similarly to human inputs.

It is sufficient to test that the most important and critical action flows of users go through without errors and that BCs call each other with right operations and parameters. Exception cases [21] are tested after normal cases. While considering exceptions an entity BC can send events to a process BC. This means that forming dependency graph needs to be slightly modified. The contracts of components specify what the components offer, not what the users need and not the order of users actions, thus contracts of BCs are not useful while testing BCS. At last, BCS's external interface is tested with local calls and calls over network. Here all the contracts of BCS must be tested.

In conclusion, before testing the whole business component system each business component in it is tested. Before a business component is tested each distributed component in it is tested. The dependencies considered stay all the time at one component granularity level.

5. RELATED WORK

According to Weyuker's [24] anticomposition axiom adequately testing each individual component in isolation does not necessarily suffice to adequately test the entire program. Interaction cannot be tested in isolation. So, it is important to test components' interaction in the new environment as components are grouped together.

Our work has got influence from Wu et al. [28]. However, we wanted that dependencies stay at the same abstraction level, i.e. they must not go from upper level to the lower level or vice versa in testing. In presentation of Wu et al. there is no clear separation of abstraction levels. For example, a dependency between components causes dependencies between classes. The interface implementation and functional logic are tested separately and not in order. In our algorithm, dependencies stay at the same granularity component level: in system level, in business component level, or in distributed component level. The dependencies between classes in object oriented systems need to be considered only at the lowest level. This reduces the dependencies and especially those dependencies, which must be considered at the same time.

Regnell et al. have considered use case modeling and scenarios in usage-based testing [17]. They investigate usage of software services in different users and users' subtypes. They consider dependency relationships, where relationships go from component level to the class and object level as in [28].

Gao et al. have presented Java-framework and tracking model to allow engineers to add tracking capability into components in component-based software [3]. The method is used for monitoring and checking various behaviors, status performance, and interactions of components. It seems that the results of Gao et al. could be added in our approach in order to support the debugging aspects.

Our research emphasizes testing functional requirements of business component system. The quality requirements of stakeholders such as security, response times and maintainability must be tested too. This has been considered in [1]. Different quality properties need to be tested separately although scenarios can be utilized here too. Testing quality requirements leads to the consideration of architectures.

6. CONCLUSION

We have presented a method for testing functionality of business component systems. For testing functionality we utilize test cases and dependency graphs. Test cases are derived from use cases or contracts.

Why do we need test cases and dependency graphs? To assure that the whole composition component's functionality has been covered by test cases it is necessary to use the dependency graphs. If we only test that the test cases are executed right, they give right result, and leave the system in consistent state, there may remain some critical paths in the system that have not been executed or there may be some paths that have been tested many times. If there are paths, which are not traversed at all our test suite does not correspond the functionality of the system. In this case, we must examine carefully if

- new test cases should be inserted or
- the components on non-traversed path are needless for some reason.

In our method, components of different granularities are tested level by level. Thus in integration testing the dependencies stay inside a business component system at the business component level. While testing business components the dependencies stay at the distributed component level. At distributed component level we consider the dependencies between classes. From the above follows that dependencies stay simple and at the same level, and the dependencies tested at the same time are similar, except at the DC level. Thus testing work is divided into small pieces and the amount of testing work decreases. This facilitates regression testing too.

Our work has been done at the University of Kuopio as a part of PlugIT research project in which our testing method will be evaluated in practice. The validation of the method containing also theoretical proof of decreasing the work in testing in practice is going on at the moment. The goal of PlugIT project is to reduce threshold of introduction of health care information systems by defining more effective and open standard solutions for system level integration. Our concern is for the quality assurance and testing of health care information systems.

7. ACKNOWLEDGMENTS

We would like to thank Hannele Toroi, testing manager at Deio for giving us insight into test implementation and testing problems in practice. This work is part of PlugIT project, which is funded by the National Technology Agency of Finland, TEKES together with a consortium of software companies and hospitals.

8. REFERENCES

[1] Bosch, J. Design and use of software architectures. Adopting and evolving a product-line approach. Addison-Wesley, 2000.

[2] Fowler, M., and Kendall, S. UML Distilled Applying the standard Object Modeling Language. Addison-Wesley, 1997.

[3] Gao, J., Zhu, E., Shim, S., and Chang, L.: Monitoring software components and component-based software. In Proc. of 24th Annual International Computer Software & Applications Conference, 2000.

[4] Gotel, O. Contribution structures for requirements traceability. PhD thesis, University of London, 1995. http://www.soi.city.ac.uk/~olly/work.html.

[5] Herzum P., and Sims, O. Business Component Factory. Wiley Computer Publishing, New York, 2000.

[6] Jacobson, Christerson, M., Jonsson, P., and Övergaard, G. Object-Oriented Software Engineering – A Use Case Driven Approach. Addison-Wesley, Harlow, 1995 2nd edn.

[7] Kaner, C. Testing computer software. John Wiley & Sons, New York, 1999.

[8] Korpela, M., Eerola, A., Mursu, A., and Soriyan, HA.: Use cases as actions within activities: Bridging the gap between information systems development and software engineering.

Abstract. In 2nd Nordic-Baltic Conference on Activity Theory and Sociocultural Research, Ronneby, Sweden, 7-9 September 2001.

[9] Kruchten, P. The Rational Unified process, an introduction. Addison-Wesley, 2001.

[10] Kung, D., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., and Chen C.: Change impact identification in object oriented maintenance. in Proc of IEEE International Conference on Software Maintenance 1994, 202-211.

[11] Kung, D., Hsia, P., and Gao, J. Testing object-oriented software. IEEE computer society, USA, 1998.

[12] Kung, D., Lu, Y., Venugopalan, N., Hsia, P., Toyoshima, Y., Chen C., and Gao, J.: Object state testing and fault analysis for reliable software systems. In Proc. of 7th International Symposium on Software Reliability Engineering, 1996.

[13] Meyer, B. Object-oriented software construction. Prentice Hall, London, 1988.

[14] Mowbray, T., and Ruh, W. Inside CORBA: Distributed object standards and applications. Addison-Wesley, 1997.

[15] Myers, G. The art of software testing. John Wiley & Sons, New York, 1979.

[16] Perry, D., and Kaiser, G.: Adequate testing and object oriented programming. Journal of Object-Oriented Programming, Jan/Feb, 1990, 13-19.

[17] Regnell, B., Runeson, P., and Wohlin, C.: Towards integration of use case modelling and usage-based testing. The Journal of Systems and Software, 50, 2000, 117-130.

[18] Robertson, S., and Robertson, J. Mastering the requirements process. Addison-Wesley, 1999.

[19] Roper, M. Software testing. McGraw-Hill, England, 1994.

[20] Sametinger, J. Software Engineering with Reusable Components. Springer-Verlag, 1997.

[21] Schneider, G., and Winters, J.P. Applying use cases. Addison-Wesley Longman, 1998.

[22] Szyperski, C. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Harlow, 1999.

[23] Turner, C.D., and Robson, D.J.: The state-based testing of object-oriented programs. In Proc. of IEEE Conference on Software Maintenance 1993, 302-310.

[24] Weyuker, E.: The evaluation of program-based software test data adequacy criteria. Communications of the ACM, 31:6, June 1988, 668-675.

[25] Wilde, N., and Huitt, R.: Maintenance Support for Object-Oriented Programs. IEEE Transactions on Software Engineering, 18, 12, Dec. 1992, 1038-1044.

[26] Vitharana, P., and Jain, H.: Research issues in testing business components. Information & Management, 37, 2000, 297-309.

[27] Wu, Y., Pan, D., and Chen, M-H.: Techniques for testing component-based software. Technical Report TR00-02, State University of New York at Albany, 2000.

[28] Wu, Y., Pan, D. and Chen, M-H. Techniques of maintaining evolving component-based software. In Proceedings of the International Conference on Software Maintenance, San Jose, CA (USA), October 2000