Incremental Class Testing from a Class Test Order

Yvan Labiche

Software Quality Engineering Laboratory
Systems and Computer Engineering - Carleton University
1125 Colonel By Drive, Ottawa ON K1S5B6, Canada
labiche@sce.carleton.ca

Abstract

Many approaches exist to decide the order in which classes should be integrated during (integration) testing. Most of them, based on an analysis of class dependencies (for instance described in a UML class diagram) aim at producing a partial order indicating which classes should be tested in sequence and which ones can be tested in parallel. We argue in this article that, thanks to the specifics of such a class test order, it is possible to define an incremental strategy for testing classes that promotes reuse during testing, not only along class inheritance hierarchies.

1 Introduction

One important problem when integrating and testing Object-Oriented (OO) software is to decide the order of class integration. As pointed out by many authors (e.g., [9]), no simple traditional technique such as top-down or bottom-up (or even sandwich) can be applied to an OO context. Indeed, most of the time, because of the many class dependencies likely introducing cycles, there is no “top” or “bottom” in class dependency graphs to start with during integration testing. Several approaches based on an analysis of class dependencies exist to solve this problem. They entail the breaking of some class dependencies (for testing purposes only) and all lead to the definition of a partial order for class testing. This class test order indicates under the form of a graph [10] which classes have to be tested in sequence (for instance to reduce testing costs) and which classes can be tested in parallel (e.g., by different testers). Then, a tester(s) can typically start from the top of the graph, i.e., from classes that do not depend on other classes, and follow the edges towards the bottom of the graph to test classes step by step.

When looking at such a graph, it appears that at each step, the target class under test is the only new class to be integrated from previous steps: the class is unit tested and its interactions with previously tested classes are exercised (integration testing). This suggests that the class test order graph can be used to incrementally test classes, step by step adding one new class at a time to the set of tested classes, and step by step exercising class interactions. In this article, we describe the interesting characteristics of such class test orders and argue, thanks to an industrial case study, that they lead to the definition of an incremental class testing strategy that promotes reuse.

Section 2 reports on relevant related work on class test orders and class (integration) testing. Section 3 then details a class test order that accounts for both static and dynamic class dependencies and presents its main properties. Sections 4 reports on an industrial case study, showing how to take advantage of those properties during (integration) testing. Conclusions are drawn in Section 5.

2 Related Work

2.1 Ordering Class Testing

A number of papers have provided strategies and algorithms to derive an integration test order from class dependencies (e.g., a UML class diagram). They are either based on graph algorithms (see a comparison of techniques in [5]), or on optimization techniques (e.g., Genetic Algorithms) [4]. The objective of all these approaches is to minimize the number of test stubs to be produced (or their complexity [4]), as this is perceived to be a major cost factor of integration testing [9]. Stubs are software units that are necessary to run the software under test and that must be developed as part of the test harness along with drivers and oracles [2]. Such a need stems from the fact that, in most software development projects, classes are developed and tested concurrently by different developers and integration begins before the class development and testing phase are complete.

Other integration strategies exist. They are not based on class dependencies, as derived during software design or reverse-engineered. They rather
associate a functional description with (a set of) classes. For instance [8], Atomic System Functions (ASF), which involve system inputs and outputs and exercise Method/Message paths between objects, drive the integration test of classes. ASFs correspond to a functional decomposition of the system which is similar to use cases. The objective of the strategy is not to minimize test stubs but to execute complete, end-user functionalities, in an incremental manner during integration. Similar strategies based on use cases can be found [3, 11].

2.2 Incremental Class Testing

Term incremental class testing has been used to describe the test of inheritance hierarchies, following the pioneer work of Harrold et al. [7]. The approach helps the tester to decide which operations of a child class have to be tested, once the parent class has passed its tests. A new or redefined operation is to be tested. (In the latter case, one can foresee that test cases defined for testing the operation in the parent class can be reused for testing the redefined operation in the child class.) An inherited operation (no modification) is retested only if it interacts (i.e., calls) with operations that are redefined in the child class. Since this pioneer work, these principles have been widely used: for instance, for operation sequences [1], for inter-class testing [13], for functional testing [16].

To the best of our knowledge, this has been the only use of term incremental testing (incremental is associated with inheritance) and no work has looked at incrementally testing classes once a class test order is available.

3 A Class Test Order and its Properties

In this section we shortly introduce the test order approach described in [10] and then discuss properties of the test order graphs it produces and how these properties are interesting for incremental class testing.

3.1 A Comprehensive Class Test Order

The class test order strategy suggested in [10], known as TOONS, relies on several principles. First, both static and dynamic dependencies between classes are analyzed. Static dependencies are those implicitly described in the class diagram (e.g., an association between two classes), whereas dynamic relationships are due to associations to parent classes, resulting in potential dependencies to the child classes because of dynamic binding.

Second, testing levels focusing on the test of individual classes are defined and a partial ordering relation is proposed to order them. The result is a directed graph indicating which testing levels (i.e., which classes) have to be tested in sequence and which ones can be tested in parallel. The main objective is to reduce to the maximum extent possible the number of testing stubs (Section 2.1). As a result, a class is tested after the classes it depends on: i.e., a client class after a server class, a child class after a parent class. Another objective of the partial order is to test dynamic dependencies after static dependencies: Once we are confident that client class A works with server class B, we verify that A can also work with descendants of B.

The third principle consists in defining testing levels in terms of: information about the class under test at each level; the set of classes involved in the level (they are necessary to perform the test and must have been tested beforehand); and the kind of dependencies taken into account (static or dynamic). The objective is to facilitate the definition of test criteria to be associated with each testing level. This is achieved by means of writing conventions for testing levels. They indicate: (1) the class under test at the testing level, which name is appended with character ‘#’; (2) the classes that need to be considered for the test and either need to be instantiated since they are used (e.g., association) by the class under test (their name appear as is) or do not need to be instantiated since they are parent (their name is shown between parentheses); (3) whether the testing level targets static or dynamic dependencies, the latter being indicated with character ‘*’ appending the parent class name that introduces the dynamic dependencies.

Finally, the test order can be modified to account, for instance, for abstract classes.

Those principles have been formalized and implemented in a software tool (a plug-in to Rational Rose) and are not further described in this article: The interested reader is referred to [10] and Appendix A.

Fig. 1 illustrates this approach on an abstract example: The UML class diagram in Fig. 1(a) shows inheritance relationships (e.g., between classes C and D), associations (e.g., between F and G) and compositions (e.g., between B and A). The class test order contains ten testing levels as shown in Fig. 1(b). Following the writing conventions, testing level “A,B#C,(D)” indicates the test of the static dependencies of B, for which A and C have to be instantiated (class D is simply a parent class). Testing level “(C),(D),F#,G*,H” indicates the test of the dynamic dependencies of class F (descendant of C and D) and G introduces dynamic dependencies as it is at the same time used by F and the parent class of H.

The class test order graph in Fig. 1(b) indicates that, in order to reduce the number of stubs to the maximum extent possible (in fact to zero in this case), the tester
has to start testing A and D (in whatever order since there is no dependency between the two testing levels). Once D is tested, the tester can proceed with C, and then with G and B (in parallel)…

![Diagram](image)

**Fig. 1. Example of class diagram (a) and class test order (b)**

Note that this strategy has originally been defined for acyclic class dependency graphs (e.g., no bidirectional associations in the UML class diagram), and that recent approaches have been proposed to address the problem of cyclic class dependency graphs: [4, 5]. They entail the breaking of some class dependencies to obtain an acyclic graph, thus suggesting that the two kinds of approaches (cycle breaking as in [4, 5] and test order as in [10]) can be combined to obtain a class test order: defining and ordering testing levels from the acyclic dependency graph obtained once cycles have been broken.

### 3.2 Important Properties of the Order

This class test order has two important properties that are direct consequences of the formally defined partial ordering [10] (proofs are not provided here due to space constraints but can be found in Appendix B):

- A testing level focusing on static dependencies adds only one class (i.e., the one under test) to the set of classes considered in previous testing levels, i.e., testing levels higher up in the order. Moreover, when considering two successive testing levels focusing on static dependencies, there is either an inheritance or association relationship (or both) between the two classes under test at each level.

For instance (Fig. 1(b)), testing level “A,B#,C,(D)” adds only class B to the classes previously tested, namely A, D and C in testing levels “A#”, “D#” and “C#,(D)”, respectively. Testing level “C#,(D)” follows testing level “D#” where C is a child class of D, and “A,B#,C,(D)” follows testing levels “C#,(D)” and “A#” where class B is associated with classes C and A.

- A testing level focusing on dynamic dependencies does not add any new class to the set of classes involved in previous levels. In other words, all the classes involved in such a level have already been tested (static dependencies) in another testing level.

For instance (Fig. 1(b)), each class involved in testing level “(C),(D),F#,G*,H” is the target of a testing level higher-up in the order.

In other words, a testing level adds at the most one new class to the set of classes considered in previous levels, thus suggesting a possible incremental testing approach for classes according to the test order graph. Indeed, once a test order has been devised, classes can be tested one at a time and this will hopefully result in a high level of reuse of test cases and test scaffolding.

### 4 An Industrial Case Study

In this section, we first shortly introduce the case study (Section 4.1) and then we show on a couple of representative examples how the order facilitates class incremental testing (Section 4.2).

#### 4.1 The ADS Case Study

The Automatic Dependent Surveillance (ADS) is a method of retrieving information about an aircraft's position and intent directly from the aircraft itself (its onboard computers), without any intervention of the crew [14]. The ADS supplies aircraft information (under the form of reports) to ground-based applications by means of three types of contracts: In a periodic contract the report is sent by the aircraft on a regular basis (specific period); In an event contract the report is sent when a specific event occurs (e.g., the aircraft arrives at a given position, its altitude passes a threshold); In a demand contract, a report is sent immediately on request.

The case study is a partial implementation of the ADS performed only in the framework of R&D activities (no deployment in actual aircrafts). The analysis and design phases follow the OMT methodology, and the programming language used is C++. The part of the design class diagram of interest in this article consists of sixteen classes (Fig. 2): classes ObjectID and Event are abstract. Event is the parent class for the events that can be observed: VertRateChg (ascending and descending rate), AltRangeChg (altitude change), LatDevChg (lateral deviation), WaypointChg (a change in the route). Class Periodic is used for each periodic contract (an
immediate contract is a periodic contract with a period of 0) and Contract represents any contract. BL_List is a bidirectional linked list and DO_List is made of two BL_List instances: one list contains information about active contracts and the other contains free memory slots to be used for new contracts. This is to address the requirements in the avionic domain that dynamic memory allocation should be avoided [15]. The control classes are in charge of controlling the events and contracts. The dependencies simply indicate that the three different lists can contain only specific types: e.g., class EventsList cannot contain instances of Contract or Periodic.

Fig. 2. Class diagram (excerpt) for the ADS

This class diagram exhibits simple static dependencies but its complexity comes from dynamic dependencies. The different subclasses of DO_List (and thus the control classes) can dynamically depend on all ObjectId’s subclasses.

4.2 Incremental testing

According to these class dependencies, 24 testing levels are defined following the TOONS approach: see Fig. 3. (The writing notation is a simplification of the convention described in [10] and only shows the class under test at each level, whose name is appended with character ‘*’ when the level targets dynamic dependencies.) Following the TOONS approach, the class test order has to be simplified (simplified graph not shown) because of abstract classes ObjectId and Event (see [10] for details on the simplifications).

The following subsections present results on incrementally testing a representative subset of those classes. They are: Class ObjectId and its subclasses (Section 4.2.1); Class BL_List (Section 4.2.2); Classes DO_List and EventsList (Section 4.2.3).

4.2.1 Testing ObjectId and its subclasses

Since class ObjectId is abstract it cannot be tested alone (as suggested in Fig. 3) and its operations are tested in the context of its child classes (this is part of the transformations of the order described in [10]). When looking at the design documents, one discovers that ObjectId’s child classes do not have any state base behavior and only hold data to be used by their clients (i.e., the list and control classes). The test of these classes, following Harrold’s approach [7], does not present any difficulty. A lot of the test scaffolding is reused (approximately 90% of the driver and oracle is reused when going from a parent class to a child class), except for one operation which specification changes in the inheritance hierarchy.

4.2.2 Testing BL_List

The test order (Fig. 3) suggests that BL_List be tested with ObjectId only (the test of BL_List directly follows the test of ObjectId), once this latter class has been tested. Since ObjectId is not tested alone (it is abstract) but along with its child classes, BL_List cannot be tested as suggested in the original test order. A simplification is required [10] that moves BL_List’s test after the test of a non-abstract child class of ObjectId, say Contract. BL_List provides the usual operations to add/remove an element to/from the beginning/end of the list. Testing this class requires testing its operations and their interactions, leading to the identification of a couple of instantiations of the association between BL_List and ObjectId (in fact, Contract). Since the association is one-to-many (one instance of BL_List linked to 0 or more instances of subclasses of ObjectId), four instantiations are required, following an approach similar to boundary value analysis and special values testing [12]. For a one-to-many association between classes X and Y we consider: One instance of X linked with zero instance of Y; One instance of X linked with one instance of Y; One instance of X linked with a maximum number (to be identified) of Y; One instance of X linked with an average number (between 1 and the identified maximum) of instances of Y. This leads to a number of test cases for the test of the static dependencies of BL_List, and some scaffolding (driver, no stub, oracle). A typical test case example is the following: add obj1 at index 0, add obj2 at index 0, add obj3 at index 1, add obj4 at index 3, remove obj5 (does not exist in the list), remove obj1 (obj1 to obj5 being instances of Contract).

When it comes to the test of the dynamic dependencies of BL_List (testing level “BL_List *” in Fig. 3), once subclasses of ObjectId have passed the test of their static dependencies, one has to

---

1 We do not describe any of the criteria we used to derive test cases as our objective is rather to show how the test order led to incrementally testing classes in the ADS.
consider all the possible subclasses of ObjectID. This has also been called the *all bindings* criterion [6]. The test cases derived for the static dependencies of BL_List can be considered as generic, the parameter being the sequence of instances of subclasses of ObjectID to use in the test cases (obj1 to obj5 in the example above). It is then very easy to execute those test cases with only instances of Periodic, or any of the subclasses of Event. Then a mix of these instances can be used: the sequence of classes to be instantiated can be determined randomly (e.g., Contract, WaypointChg, Periodic, VertRateChg and LatDevChg for obj1 to obj5 above, respectively). This requires only minor modifications (mostly copy-paste) to the driver and oracle used for the test of the static dependencies of BL_List because its behavior does not depend on the types of the instances it holds.

By separating the test of static and dynamic dependencies, the test order led to a very high level of reuse of the scaffolding: the driver and oracle built for testing level “BL_List” was reused, almost entirely, for testing level “BL_List *”.

### 4.2.3 Testing DO_List and EventsList

The test of DO_List followed similar principles. First its static dependencies are tested with two instances of BL_List (see the multiplicities in Fig. 2) and instances of Contract (again, ObjectID cannot be used). Some reuse of the scaffolding for the test of BL_List was possible only because BL_List and DO_List have common functionalities: addition, deletion of elements in the list(s). Additional test cases were required because of functionalities specific to DO_List which handles a list of active contracts and a list of available memory slots\(^2\). Then the test of the dynamic dependencies of DO_List also considered two instances of BL_List and different instantiations and bindings to child classes of ObjectID. Similarly to BL_List, almost all the scaffolding required for the test of DO_List’s static dependencies was reused for the test of its dynamic dependencies.

Class EventsList has three new operations compared to DO_List, requiring new test cases. These new operations bear some similarities with operations defined in DO_List: For instance, an operation in DO_List removes an instance of ObjectID’s subclass according to its reference whereas an operation in EventsList does the same but from the unique identifier of the instance (attribute in ObjectID). This led to the reuse of test cases defined for DO_List in which operations newly defined in EventsList replace operations defined in DO_List (simple copy-paste in the driver). More specifically, only 12% of the driver for EventsList is new from the driver for DO_List, and the oracle for the two classes is almost the same. This is however a situation specific to classes DO_List and EventsList that should not be expected in every inheritance hierarchy. Once again, the test cases and scaffolding required for the test of the static dependencies of EventsList is reused almost entirely for the test of its dynamic dependencies. The only difference here is that instead of any instance of any subclass of ObjectID, only subclasses of Event are considered.

---

\(^2\) Note that the design could have described DO_List as a child class of BL_List.
5 Conclusion

Traditional (procedural) integration testing strategies are not adequate for object-oriented systems and there exist several approaches to produce class integration strategies from class diagrams. These strategies all lead to deriving a partial order (represented as a graph) indicating which classes have to be tested in sequence and which ones can be tested in parallel.

In this article we have shown that, thanks to the specific properties of such a class test order, it is possible to incrementally test classes. Incremental class testing then does not only mean testing inheritance hierarchies, but also testing client-server relationships incrementally. In this context, incrementally also means testing static dependencies before dynamic ones and saving effort by reusing a lot of the scaffolding required for class (integration) testing.

Although this article has investigated the problem on one industrial case study, and more results are required, we believe that a class test order is an interesting roadmap that can facilitate (promote?) reuse.

Another opportunity for reuse occurs when one has to test the dependencies that were broken in order to obtain an acyclic class dependency graph and order testing levels. This time, instead of stubs simulating the target classes of the broken dependencies, the actual classes are used and the test cases used for testing the client classes in those dependencies have to be rerun. Possible challenges will be investigated in future work.

Note that such a roadmap can also be used for the implementation of the classes once the design is considered (sufficiently) complete. Another possible use is to help plan and schedule testing (or implementation) once the tester (designer) has a good idea of the effort required for each step in the order. This however requires further research in terms of testing/implementation effort measurement in an object-oriented context.

6 Acknowledgements

This work was performed within the framework of the Laboratory for Dependability Engineering (LIS), a Cooperative Laboratory between leading industrial organizations and research laboratory LAAS-CNRS. Yvan Labiche is further supported by an NSERC operational grant.

7 References


Appendix A  Formalism

A.1 Static and Dynamic Dependencies

The determination of testing levels needs information about classes and their static as well as dynamic dependencies. Such information is provided by three functions that compute, for each class X:

$D_1(X)$: The set of classes on which X depends statically—either directly or by transitivity according to the class diagram. $D_1(X)$ is the minimum set of classes required at compile time to test class X;

$D_2(X)$: The set of classes on which X depends either statically or dynamically (i.e., at execution time) or both—either directly or by transitivity. $D_2(X)$ is a superset of $D_1(X)$ that takes polymorphism into account, if any. In addition to $D_1(X)$, it includes all the child classes of the classes that are used (directly or by transitivity). $D_2(X)$ is the maximum set of classes required to test class X.

$B_d(X)$: A Boolean function that indicates whether or not X may dynamically depend on at least one class of $D_2(X)$, due to polymorphism. $B_d(X)$ has value true (or simply 1) if at least one polymorph relationship is identified and false (or simply 0) otherwise.

Table 1 below shows the result of these three functions for the classes in Fig. 4. Class A does not depend on any other class ($D_1(A) = D_2(A) = \emptyset$ and $B_d(A) = 0$). Class C only depends on class D and does not have any dynamic dependency ($D_1(C) = D_2(C) = \{D\}$ and $B_d(C) = 0$; it is not associated, directly or by transitivity, with a parent class). Class F statically depends on classes C, D, and G, and dynamically depends on class H in addition to the three previous classes: $B_d(F)= 1$, $D_1(F)= \{C, D, G\}$, and $D_2(F)= \{C, D, G, H\}$ (since F depends through an association on class G, which is a parent class for H, F can depend at runtime on H).

![Fig. 4. An example class diagram](image)

More formally, given $Cl$ the set of all classes of interest, let us define the three following relations:

- $\text{inherits}_\text{from}$: Identifies classes related through the generalization relationship.
  
  $\text{inherits}_\text{from} = \{<X,Y>/ X\in Cl, Y\in Cl, and X \text{ inherits from } Y\}$

- $\text{uses}$: Identifies class client/server relationships (e.g., associations, dependencies).
  
  $\text{uses} = \{<X,Y>/ X\in Cl, Y\in Cl, and X \text{ is a client of } Y\}$

- $R_s$: Identifies direct class dependencies.
  
  $R_s = \text{inherits}_\text{from} \cup \text{uses}$

- $\text{inherited}_\text{by}$: Identifies ancestor classes.
  
  $\text{inherited}_\text{by} = (\text{inherits}_\text{from}^+)^{-1}$

- $R_d$: Identifies direct dependencies due to polymorphism, i.e., dependencies to child classes of used parent classes.
  
  $R_d = \text{inherited}_\text{by} \circ \text{uses}$

- $\text{uses}^*$: Denotes the reflexive transitive closure of $\text{uses}$.

Then, for any class X, $D_1(X)$, i.e., the set of classes that X depends on statically (direct dependencies or through transitive closure), can be defined with $R_s$ as follows:

$\forall X\in Cl, D_1(X) = \{Y\in Cl / <X,Y>\in R_s^+\}$, where $R_s^+$ is the transitive closure of $R_s$.

Similarly, for any class X, $D_2(X)$, i.e., the set of classes that X depends on dynamically (direct dependencies or through transitive closure), can be defined with $R_s$ and $R_d$ as follows:

$\forall X\in Cl, D_2(X) = \{Y\in Cl / <X,Y>\in (R_s \cup R_d)^+\}$

Last, $B_d(X)$ can also be defined from $R_s$ and $R_d$:

$\forall X\in Cl, B_d(X) = 1 \iff \{Y/ <X,Y> \in R_s^*(R_s \cup R_d)^+\} \neq \emptyset$

(where * denotes the reflexive transitive closure).

In other words, $(R_s \cup R_d)^+$ denotes the set of classes that can be reached from X using paths of any length involving both static and dynamic dependencies. Then, $R_s^*(R_s \cup R_d)^+$ retains only the classes from this set that
can be reached from $X$ using at least one dynamic dependency.

A.2 Testing levels

A testing level $T$ is represented by a triplet $(T.\text{goal}, T.\text{need}, T.\text{type})$ where: $T.\text{goal}$ is the set of classes under test in testing level $T$; $T.\text{type}$ indicates the type of dependencies taken into account: either static (Sta) or dynamic (Dyn); $T.\text{need}$ is the set of classes that have to be involved during the test of classes in $T.\text{goal}$. It contains the classes under test (T-goal) and all the classes on which they are dependent according to $T.\text{type}$. In other words, testing level $T = (T.\text{goal}, T.\text{need}, T.\text{type})$ indicates that classes in set $T.\text{need}$ are required for the test of the $T.\text{type}$ dependencies of classes in $T.\text{goal}$.

The testing levels are produced as triplets according to the following strategy:

(i) For every class $X$, a specific testing level $T_X$ is defined that accounts for static dependencies only: $T_X.\text{goal} = \{X\}$, $T_X.\text{need} = D_1(X) \cup \{X\}$, and $T_X.\text{type} = \text{Sta}$;

(ii) For every class $X$, testing of additional (i.e., dynamic) dependencies that may occur at execution time (i.e., if $B_d(X)=1$), is the target of a separate level. Two cases have to be considered, depending on whether class $X$ is involved in a cycle due to dynamic dependencies or not.

- If $X$ is not involved in such a cycle, i.e., $X \notin D_2(X)$, the following testing level is created: $T_X.\text{goal} = \{X\}$, $T_X.\text{need} = D_2(X) \cup \{X\}$, and $T_X.\text{type} = \text{Dyn}$.
- If $X$ is involved in such a cycle, i.e., $X \in D_2(X)$, the following testing level is created: $T_X.\text{goal} = \{X_1, \ldots, X_n\}$, $T_X.\text{need} = D_2(X)$, and $T_X.\text{type} = \text{Dyn}$, where $\{X_1, \ldots, X_n\}$ is the set of classes involved in the cycle.

Note that, although it is assumed that initial cycles in the class diagram have been broken (the class dependencies used to compute $D_1$, $D_2$ and $B_d$ do not contain any cycle), cycles may exist while considering dynamic dependencies in addition to static ones. For example, polymorphism introduces a cycle between classes $B$ and $E$ in Fig. 4 (B dynamically depends on $E$ because of polymorphism, which itself depends on $B$). More generally, when a class $X_i$ is involved in such a cycle, then by construction: (i) $X_i \in D_2(X_i)$, and (ii) all the classes $X_1, \ldots, X_n$ involved in a same cycle have identical sets $D_2(X_1) = \ldots = D_2(X_n)$. It results that only one testing level targeting the dynamic dependencies of all these classes is created.

Table 2 shows the ten testing levels built for the classes in Fig. 4: 8 testing levels of type Sta and 2 testing levels of type Dyn. The 8 testing levels of type Sta are straightforward from the definition above and the data in Table 1. Table 1 shows three classes requiring a testing level of type Dyn (i.e., with $B_d(X)=1$): $B$, $E$, and $F$. However, since there is a cycle due to dynamic dependencies between $B$ and $E$ ($D_2(B) = D_2(E)$), only one testing level is built with goal $\{B, E\}$.

### Table 2. Testing levels for classes in Fig. 4

<table>
<thead>
<tr>
<th>$T.\text{goal}$</th>
<th>$T.\text{need}$</th>
<th>$T.\text{type}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${A}$</td>
<td>${A}$</td>
<td>Sta</td>
</tr>
<tr>
<td>${B}$</td>
<td>${A, B, C, D}$</td>
<td>Sta</td>
</tr>
<tr>
<td>${B, E}$</td>
<td>${A, B, C, D, E, F, G, H}$</td>
<td>Dyn</td>
</tr>
<tr>
<td>${C}$</td>
<td>${C, D}$</td>
<td>Sta</td>
</tr>
<tr>
<td>${D}$</td>
<td>${D}$</td>
<td>Sta</td>
</tr>
<tr>
<td>${E}$</td>
<td>${A, B, C, D, E}$</td>
<td>Sta</td>
</tr>
<tr>
<td>${F}$</td>
<td>${C, D, F, G}$</td>
<td>Sta</td>
</tr>
<tr>
<td>${F}$</td>
<td>${C, D, F, G, H}$</td>
<td>Dyn</td>
</tr>
<tr>
<td>${G}$</td>
<td>${C, D, G}$</td>
<td>Sta</td>
</tr>
<tr>
<td>${H}$</td>
<td>${C, D, G, H}$</td>
<td>Sta</td>
</tr>
</tbody>
</table>

A.3 Ordering relation between testing levels

The partial ordering relation used to determine a test order between the testing levels is defined as follows.

Definition: Let $M = (M.\text{goal}, M.\text{need}, M.\text{type})$ and $N = (N.\text{goal}, N.\text{need}, N.\text{type})$ be two testing levels. Then,

$$M \subseteq N \iff M.\text{type} = N.\text{type} \land M.\text{need} \subseteq N.\text{need}$$

or

$$M \subset N \iff (M.\text{type} = \text{Sta} \land N.\text{type} = \text{Dyn}) \land M.\text{need} \subset N.\text{need}$$

where $M \subset N$ means that level $M$ precedes level $N$ in the class test order.

A class test order graph can be built easily from relation $\subseteq$ as follows:

- The nodes of the graph are the testing levels, targeting static and dynamic dependencies for all the classes of interest.
- For two distinct testing levels (i.e., nodes) $N_1$ and $N_2$, there is an edge from $N_1$ to $N_2$ if and only if $N_1$ precedes $N_2$ (i.e., $N_1 \prec N \land N \prec N_2$), and there is no other testing level (node) $N$ preceding $N_2$ and being preceded by $N_1$, i.e., $\neg(N_1 \prec N \land N \prec N_2)$.

Fig. 5 shows the class test order for the classes in Fig. 4. Testing level $\{\{A\}, \{A\}, \text{Sta}\}$ immediately precedes testing level $\{\{B\}, \{A, B, C, D\}, \text{Sta}\}$. Testing level $\{\{C\}, \{C, D\}, \text{Sta}\}$ is preceded by $\{\{D\}, \{D\}, \text{Sta}\}$.
and precedes ({B}, {A,B,C,D}, Sta) and ({G}, {C,D,G}, Sta).

({A}, {A}, Sta) →

({B}, {A,B,C,D}, Sta) →

({C}, {C,D}, Sta)

({D}, {D}, Sta)

({E}, {A,B,C,D,E}, Sta) →

({F}, {C,D,F,G}, Sta) →

({G}, {C,D,G}, Sta) →

({H}, {C,D,G,H}, Sta)

({F}, {C,D,F,G,H}, Dyn) →

({B,E}, {A,B,C,D,E,F,G,H}, Dyn)

Fig. 5. Class test order for Fig. 4

A.4 Writing Conventions

The triplet representation of testing level T explicitly shows which classes are the targets of testing (T.goal) and whether or not polymorphism is to be taken into account (T.type). However, this representation is poorly informative regarding to the classes involved at level T, since all we get is their set T.need without any further information. Hence, we suggest a more descriptive textual representation of the testing levels: A testing level is written as a list of class names with additional special characters denoting the role of the classes within the testing level. Table 3 shows the writing conventions associated with different possible roles. These roles may easily be deduced from the types of static relationships indicated in the class diagram, and the additional polymorph relationships identified to build the sets D2(X).

As shown in Table 3, from the role of each class (and thus, the writing conventions), it may be deduced whether or not each class has to be instantiated during testing.

Table 3—Writing conventions

<table>
<thead>
<tr>
<th>Role of class X ∈ T.Need</th>
<th>Notation</th>
<th>Instantiation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class under test (∈ T.Goal)</td>
<td>X#</td>
<td>Yes</td>
</tr>
<tr>
<td>Parent class only</td>
<td>(X)</td>
<td>No</td>
</tr>
<tr>
<td>Server class only</td>
<td>X</td>
<td>Yes</td>
</tr>
<tr>
<td>Both parent and server class:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Static level (T.type = Sta)</td>
<td>X+</td>
<td>Yes</td>
</tr>
<tr>
<td>• Dynamic level (T.type = Dyn)</td>
<td>X*</td>
<td>Yes, to act as a server class</td>
</tr>
</tbody>
</table>

Fig. 6 shows the class test order graph for the classes in Fig. 4. It is a re-writing of the order in Fig. 5 with writing conventions described in Table 3. For example (C), {C,D}, Sta) has textual representation C# (D) which makes it possible to see at a glance that: (i) F is the target of this testing level; (ii) C and D are parent classes (not instantiated); (iii) G introduces polymorphism.

Showing explicitly the role of each class facilitates the identification of infeasible testing levels (e.g., a testing level may require that an abstract class be instantiated), and gives a first insight into the expected complexity of the test experiments (for example, in terms of numbers of classes that may introduce polymorphism at a given level).

Note that writing conventions can be formalized in the same way we formalized our approach in previous sections, using relations uses, inherits_from, … Such a formalization is however not reported here.

Fig. 6. Class test order (with writing conventions) for Fig. 4
Appendix B Properties

B.1 Properties of D₁(X) and D₂(X)

Property 1: ∀ X ∈ Cl, X ∉ D₁(X)
Proof:
Self-evident since the class dependencies are assumed to be acyclic.

Property 2: ∀ X ∈ Cl, D₁(X) ⊆ D₂(X)
Proof:
Self-evident, by construction of D₁(X) and D₂(X).

Property 3: ∀ Y ∈ Cl, \( D_{1}(Y) = \bigcup_{X \subseteq D_{1}(Y)} \{X\} \)
Proof:
From the definition of D₁(), we have: X ∈ D₁(Y) ⇒ D₁(X) ⊆ D₁(Y).
Then, the union on the right part of the equality is included in the left part of the equality.
Additionally, D₁(Y) ⊇ \( \bigcup_{X \subseteq D_{1}(Y)} \{X\} \),
thus the equality.

Property 4: ∀ Y ∈ Cl, \( D_{2}(Y) = \bigcup_{X \subseteq D_{2}(Y)} \{X\} \)
Proof:
From the definition of D₂(), we have: X ∈ D₂(Y) ⇒ D₁(X) ⊆ D₂(Y).
Then, the union on the right part of the equality is included in the left part of the equality.
Following the same procedure as for Property 3, we prove the equality.

Property 5: ∀ X, Y ∈ Cl, \( D_{1}(X) ∪ \{X\} = D_{1}(Y) ∪ \{Y\} \) ⇔ X = Y
Proof of ⇐:
Self-evident.
Proof of ⇒:
Assume D₁(X) ∪ \{X\} = D₁(Y) ∪ \{Y\} and X ≠ Y. Then, since X ≠ Y, X ∈ D₁(Y) and Y ∈ D₁(X). In other words, X and Y statically depend on each other, i.e., there is a cycle in the static dependencies, which is against our hypothesis.
As a consequence, X = Y.

Property 6: ∀ X, Y ∈ Cl,
\( D_{1}(X) ∪ \{X\} = D_{1}(Y) ∪ \{Y\} \) ⇔ \( X = Y \) or \( X ≠ Y, X ∈ D_{2}(Y) \) and \( Y ∈ D_{2}(X) \)
Proof of ⇐:
Self-evident when X = Y.
Assuming X ≠ Y, X ∈ D₂(Y) and Y ∈ D₂(X) imply that there is a cycle involving static and dynamic dependencies between X and Y. In such a situation, because of the transitive closure in the definition of D₂(), D₂(X) = D₂(Y). Then, D₂(X) ∪ \{X\} = D₂(Y) ∪ \{Y\}.
Proof of ⇒:
Assuming X ≠ Y, then D₂(X) ∪ \{X\} = D₂(Y) ∪ \{Y\} implies that X ∈ D₂(Y) and Y ∈ D₂(X).

B.2 Properties of the ordering relation

B.2.1 Relation \( ≤ \) is a strict partial ordering

We show below the relation \( ≤ \) is non-reflexive, transitive, anti-symmetric, and partial, i.e., it is a strict partial ordering relation.

Non-reflexive:
Given M a testing level, we cannot have M ⊆ M since M.need ∉ M.need (part (1) of the definition of \( ≤ \)) or (M.type = Sta and M.type = Dyn) (part (2) of the definition of \( ≤ \)).

Transitivity:
Given M, N and O, three testing levels such that M ⊆ N and N ⊆ O, two situations have to be considered.
1. M and O have the same type: M.type = O.type.
   Assuming this common type is Sta, and given that N ⊆ O, N has also type Sta. Since M, N and O have the same type Sta, and given that M ⊆ N and N ⊆ O, M.need ⊆ N.need ⊆ O.need. In other words (M.type = O.type and M.need ⊆ O.need), M ⊆ O. Assuming this common type is Dyn, and given that M ⊆ N and N ⊆ O, N has necessarily type Dyn. (Otherwise, if N.type = Sta, M ⊆ N is not possible.) Given that M, N and O have the same type Dyn and that M ⊆ N and N ⊆ O, the following holds: M.need ⊆ N.need ⊆ O.need. In other words, M ⊆ O.
2. M and O do not have the same type. Then M.type = Sta and O.type = Dyn since, otherwise M ⊆ N and N ⊆ O would not hold at the same time.
   Given that M ⊆ N and N ⊆ O, either M.need ⊆ N.need or O.need (if N.type = Sta) or M.need ⊆ N.need ⊆ O.need (if N.type = Dyn) holds. That is, M ⊆ O.
Anti-symmetric:

Given M and N two testing levels such that M ⊆ N, let us show that N ⊆ M is not possible.
Assuming M and N have the same type, M.need ⊆ N.need, that is N.need ⊆ M.need. N ⊆ M is not possible. Assuming M and N do not have the same type, and given that M ⊆ N, we have M.type = Sta and N.type = Dyn. Then N ⊆ M is not possible.

Partial:

Two different testing levels may not be comparable by means of relation ⊆. This is the case for instance for the testing levels targeting the static dependencies of two different child classes of one parent class, when those two child classes do not depend on each other.

B.2.2 Other properties of the ⊆ relation

Let us first introduce some notation that we will use in this section. Assume X and Y are two distinct classes and that T and S are the testing level targeting their static dependencies, respectively: T = (\{X\}, D1(X)∪\{X\}, Sta) and S = (\{Y\}, D1(Y)∪\{Y\}, Sta). Further assume that P and O are the testing levels targeting their dynamic dependencies, respectively: P = (P.goal, D2(X)) and O = (O.goal, D2(Y)). We thus assume that X and Y do not participate in the same cycle due to dynamic dependencies, and thus that O ≠ P.

Property 7: Class X statically depends on class Y ⇔ S ⊆ P. In other words, Y ∈ D1(X) ⇒ S ⊆ P.

Proof:

By construction of D1(), X statically depends on Y implies that D1(Y) ⊆ D1(X). Since Y ∈ D1(X), D1(Y)∪\{Y\} ⊆ D1(X). Since the class static dependencies do not show any cycle, D1(X) ⊆ D1(Y)∪\{X\}. So, D1(Y)∪\{Y\} ⊆ D1(X)∪\{X\}, i.e., S.goal ⊆ T.goal. Then, S ⊆ T.

Proof of ⇐:

Assuming S ⊆ T, we have S.goal ⊆ T.goal, i.e., D1(Y)∪\{Y\} ⊆ D1(X)∪\{X\}. Since X ≠ Y, this means that Y ∈ D1(X).

Property 8: Bd(X) = 1 ⇒ T ⊆ P.

Proof:

According to Property 2, D1(X) ⊆ D2(X). So, T.goal ⊆ P.goal, which means that T ⊆ P.

Property 9: Class X dynamically depends on class Y ⇒ S ⊆ P.

In other words, Y ∈ D2(X) ⇒ S ⊆ P.

Proof:

According to Property 4, D1(Y) ⊆ D2(X). Since Y ∈ D2(X), D1(Y)∪\{Y\} ⊆ D2(X). Since D2(X) ⊆ D2(X)∪\{X\}, we obtain S.goal ⊆ P.goal, and then S ⊆ P.

Property 10: Assuming Bd(X) = 1 and Bd(Y) = 1, X dynamically depends on Y and O ≠ P ⇔ O ⊆ P. In other words, Y ∈ D2(X) and X ∈ D2(Y) ⇔ O ⊆ P.

Proof of ⇒:

According to the definition of D2(), D2(Y) ⊆ D2(X). Since X ≠ Y, D2(Y)∪\{Y\} ⊆ D2(X)∪\{X\}, that is O.goal ⊆ P.goal. O ⊆ P.

Proof of ⇐:

O ⊆ P implies that D2(Y)∪\{Y\} ⊆ D2(X)∪\{X\}. Since X ≠ Y, this means that Y ∈ D2(X) and X ∈ D2(Y).

It is worth noting that these properties satisfy the high level requirements that we identified as paramount for a class test order:
- A class is tested after the classes it depends on. This is satisfied among testing levels targeting static dependencies (Property 7) as well as among testing levels targeting dynamic dependencies (Property 10).
- Testing levels targeting dynamic dependencies are considered after testing levels targeting static dependencies (Property 8 and Property 9).

Let us use the classes in Fig. 4 and the class test order displayed in Fig. 5 (triplet representation) and Fig. 6 (with writing conventions) to illustrate these properties:
- Class B statically depends on classes A, C, D. Then testing level A B# C (D) precedes testing level A#, C# (D), and D# (Property 7).
- Class F uses parent class G and thus requires a specific testing level targeting its dynamic dependency to H. Testing level (C) (D) F# G# H thus precedes testing level (C) (D) F# G# H (Property 8). Additionally, (C) (D) G# H precedes testing level (C) (D) F# G# H (Property 9).
- Class B dynamically depends on class F and both require testing levels targeting their dynamic dependencies: Bd(B)=1, Bd(F)=1. Then testing level (C) (D) F# G* H (dynamic dependencies of F) precedes testing level A*
B.2.3 Properties of a test order (graph) based on relation $\subset$

Let us denote $I(S)$ the set of classes involved in testing levels that immediately precede testing level $S$ in the test order graph (recall Section A.3).

**Property 11**: Given $T=\langle\{X\}, T.\text{need}, \text{Sta}\rangle$ the testing level targeting the static dependencies of class $X$.

$T$ requires only one new class from the set of classes involved in immediately preceding testing levels, and this class is $X$. In other words, $T.\text{need} = I(T) \cup \{X\}$.

**Proof**: Let us first consider the set of preceding testing levels, as opposed to the set of immediately preceding testing levels.

$T=\langle\{X\}, T.\text{need}, \text{Sta}\rangle=\langle\{X\}, D(Y) \cup \{Y\}, \text{Sta}\rangle$

According to Property 3, $D(Y) \subseteq T.\text{need}$. (It is a $\subseteq$ instead of a $\subset$ since we do not know the type of testing level $T_i$.)

Then, $T.\text{need} = D(Y) \cup \{Y\} = \{Y\} \cup D(Y)$.

Since static class dependencies do not produce any cycle, $\{Y\} \not\subseteq D(Y)$. The above expression then shows that $T$ only adds class $X$ to the set of classes involved in preceding testing levels in the order.

Given that relation $\subset$ is transitive, it is then self-evident that $T$ only adds class $X$ to the set of classes involved in immediately preceding testing levels in the order.

**Property 12**: Given $T=\langle\{X\}, T.\text{need}, \text{Dyn}\rangle$ the testing level targeting the dynamic dependencies of class $X$.

$T$ does not require any new class from the set of classes involved in immediately preceding testing levels. In other words, $T.\text{need} = I(T)$.

**Proof**: Let us note $T_i (i=1, ..., n)$ the testing levels that immediately precede $T$ in the test order.

Then, $\forall i \in \{1, ..., n\}, T_i \subset T$, that is, $\forall i \in \{1, ..., n\}$, $T_i.\text{need} \subseteq T.\text{need}$. (It is a $\subseteq$ instead of a $\subset$ since we do not know the type of testing level $T_i$.)

Then, $\bigcup_{i=1,...,n} T_i.\text{need} \subseteq T.\text{need}$, that is $I(T) \subseteq T.\text{need}$.

Let us assume that this inclusion is strict. Then $\exists Y \in T.\text{need}$ such that $Y \not\in I(T)$.

Let us note $S$ the testing level targeting the static dependencies of class $Y$: $S=\langle\{Y\}, D(Y) \cup \{Y\}, \text{Sta}\rangle$.

$Y \in T.\text{need}$ means that $X$ depends statically or dynamically on $Y$. Then, according to Property 7 and Property 9, $S \subset T$.

Additionally, $S \not\subseteq \{T_1, ..., T_n\}$ (the testing levels immediately preceding $T$) since $Y \not\in I(T)$.

Since $\subset$ is transitive, $\exists j \in \{1, ..., n\}$ such that $S \subset T_j$.

Then, according to the definition of $\subset$, $D(Y) \cup \{Y\} \subseteq T_j.\text{need}$, i.e., $Y \in T_j.\text{need}$. That is $Y \in I(T)$.

We have however assumed the inverse. There is a contradiction. Our hypothesis that the inclusion $I(T) \subseteq T.\text{need}$ was strict is then wrong.

This proves that $I(T) = T.\text{need}$.

For instance, referring to the classes in Fig. 4 and the class test order displayed in Fig. 5 (triplet representation) and Fig. 6 (with writing conventions), we can see that:

- Testing level $C\# (D)$ only adds one class, i.e., $C$, to the set of classes involved in the immediately preceding testing level $D\#$.
- Testing level $A B\# C (D)$ only adds class $B$ (the target class of the testing level) to immediately preceding testing levels $A\#$ and $C\# (D)$.
- Testing level $C (D) F\# G\star H$ does not add any new class to the set of classes involved in immediately preceding testing levels $(C) (D) F\# G$ and $(C) (D) (G) H\#$. 