Dr. John D. McGregor is an associate professor of computer science at Clemson University and a Principal Partner in Software Architects, a consulting firm that specializes in object technology. With David A. Sykes, he is author of *Object-Oriented Software Development: Engineering Software for Reuse* published by Van Nostrand Reinhold. He is also author of numerous research and technical papers on object-oriented software practices. He is an international consultant to industry and is active in ACM and IEEE CS. For Software Architects, he conducts research on testing object-oriented software systems and conducts classes on design and testing of object-oriented software.

Dr. Timothy D. Korson is Director of COMSOFT, a research associate at Clemson University, and a Principal Partner in Software Architects. He is an international consultant to industry and author of numerous technical papers on object technology. He conducts research on metrics and management strategies for object-oriented software projects.
<table>
<thead>
<tr>
<th>Class</th>
<th>Iteration 1</th>
<th>Iteration 2</th>
<th>Iteration 3</th>
<th>Iteration 4</th>
<th>Iteration 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class 1</td>
<td>32</td>
<td>26</td>
<td>19</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>Class 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Class 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Errors</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 2: Error Rates Per Class and Per Iteration**
<table>
<thead>
<tr>
<th>Test Suites</th>
<th>Development Products</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CRC Cards</td>
</tr>
<tr>
<td>Class-level</td>
<td></td>
</tr>
<tr>
<td>Functional</td>
<td></td>
</tr>
<tr>
<td>Class-level</td>
<td></td>
</tr>
<tr>
<td>Structural</td>
<td></td>
</tr>
<tr>
<td>Class-level</td>
<td>X</td>
</tr>
<tr>
<td>Interaction</td>
<td></td>
</tr>
<tr>
<td>System</td>
<td>X</td>
</tr>
</tbody>
</table>

**Table 1: Development Products as Sources of Test Cases**
Contract DetectorController
Class Detector will provide:
   StatusCode ShowStatus() by: May 15
   void Reset() by: June 1

Class Controller will provide:
   void NotifyOfArrival(Detector) by: May 20

...
Figure 7
Figure 6
Figure 5
Figure 4
Figure 3
Figure 2
Side-bar on Flow of SASY Activities

Domain Analysis
Application Analysis
Architectural Design
Class and Cluster Development
Incremental Integration
System Testing
## Side-bar on the Relationships between SASY Work Products and the SASY Development Process

<table>
<thead>
<tr>
<th>Specific SASY Work Product</th>
<th>Relationships to Other Products and to the Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified CRC Card</td>
<td>A core product in Domain Analysis, but can be created anytime the need for a new class arises; CRC cards are a precursor to class specifications and input to the process of building object models.</td>
</tr>
<tr>
<td>Class Specification</td>
<td>A transformation and elaboration of CRC cards, responsibilities map to methods. Classes are specified at this level during Application Design.</td>
</tr>
<tr>
<td>Class Stub</td>
<td>A first pass implementation of the class specification. This evolves into the class implementation during the class development phase.</td>
</tr>
<tr>
<td>Class Implementation</td>
<td>Elaboration of class stubs done during class development.</td>
</tr>
<tr>
<td>Rumbaugh Object Model</td>
<td>A core product across all the phases. The object model starts during domain analysis as un-annotated relationships between informally specified classes. The object model evolves to rigorously annotated relationships between formally specified classes. Application Analysis adds software objects and relationships. Later phases add design mechanisms and implementation detail to the analysis models.</td>
</tr>
<tr>
<td>Harrell State Charts</td>
<td>Classes that participate in the object models along with clusters and subsystems that participate in the architecture may have their dynamic behavior modeled with state charts. Consistency between the models is an important issue. State information is modeled as needed. Domain standards specify the dynamic behavior of certain classes. This information is captured during domain analysis. Other state charts may not be developed until design.</td>
</tr>
<tr>
<td>Interaction Diagrams</td>
<td>Like state charts, interaction diagrams are developed as needed. These functional models integrate individual class, clusters and subsystems.</td>
</tr>
<tr>
<td>Clusters, subsystems, and architectures</td>
<td>While primarily a design activity, clusters and standard architectures may be identified during Domain or Application Analysis.</td>
</tr>
<tr>
<td>Modified Use Cases</td>
<td>As transformation of the specification model, use cases can be used both to test the specification for accuracy and to test object models for completeness. Use cases are initially developed during early iterations of Domain and Application Analysis, but should be incrementally expanded as each successive system increment is built.</td>
</tr>
<tr>
<td>Specification text and diagrams</td>
<td>Contrary to most traditional processes, a detailed system specification is not the first product developed. Domain models are developed first, detailed specifications are developed during Application Analysis.</td>
</tr>
</tbody>
</table>
## Side-bar on Classifying SASY Work Products

<table>
<thead>
<tr>
<th>Type of Work Product</th>
<th>Type of Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Static</td>
</tr>
<tr>
<td>Class</td>
<td>• Modified CRC Card</td>
</tr>
<tr>
<td></td>
<td>• Class Specification</td>
</tr>
<tr>
<td></td>
<td>• Class Stub</td>
</tr>
<tr>
<td></td>
<td>• Class Implementation</td>
</tr>
<tr>
<td>Cluster</td>
<td>• Rumbaugh Object Model</td>
</tr>
<tr>
<td></td>
<td>• Cluster Patterns</td>
</tr>
<tr>
<td></td>
<td>and Frameworks</td>
</tr>
<tr>
<td>Subsystem</td>
<td>• Rumbaugh Object Model</td>
</tr>
<tr>
<td></td>
<td>• Subsystem Architecture</td>
</tr>
<tr>
<td>System</td>
<td>• Rumbaugh Object Model</td>
</tr>
<tr>
<td></td>
<td>• System Architecture</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Side-bar on Selected SASY Work Products

<table>
<thead>
<tr>
<th>SASY Work Product</th>
<th>Work product description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>English text description of a single class, its responsibilities and its client relationships with other classes.</td>
</tr>
<tr>
<td>CRC Card</td>
<td></td>
</tr>
<tr>
<td>Class Specification</td>
<td>Complete specification of a class including method names, number and type of parameters, return values, pre and post conditions and class invariants along with standard information such as revision history, etc.</td>
</tr>
<tr>
<td>Class Stub</td>
<td>“Header file” completely coded with the implementation stubbed in.</td>
</tr>
<tr>
<td>Class Implementation</td>
<td>Completely coded and tested class.</td>
</tr>
<tr>
<td>Rumbaugh Object Model</td>
<td>Shows the static structural relationships (association, aggregation, classification) between classes in the model.</td>
</tr>
<tr>
<td>Harel State Charts</td>
<td>Standard state transition diagrams for classes, clusters, and subsystems.</td>
</tr>
<tr>
<td>Interaction Diagrams</td>
<td>A model of the important algorithms that span objects, clusters, or subsystems.</td>
</tr>
<tr>
<td>Clusters, subsystems, and architectures</td>
<td>Sets of related classes and the relationships between these sets</td>
</tr>
<tr>
<td>Modified Use Cases</td>
<td>A transformation of the specification model into a structured hierarchy of system usages.</td>
</tr>
<tr>
<td>Text and diagrams</td>
<td>The specification is a model of the real system requirements as they exist in the business world.</td>
</tr>
</tbody>
</table>
## Side-bar on SASY Activities

<table>
<thead>
<tr>
<th>Activity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain Analysis</td>
<td>Models of bodies of knowledge are constructed. The scope of these areas are only loosely constrained by the purpose of the application to be developed.</td>
</tr>
<tr>
<td>Application Analysis</td>
<td>The domain models and specific system requirements are used to produce a set of models that define the application to be built.</td>
</tr>
<tr>
<td>Architectural Design</td>
<td>A skeleton outline of the system is created. Utilizing standard frameworks for the various subsystems, interfaces for the subsystem interactions are defined.</td>
</tr>
<tr>
<td>Class &amp; Cluster Development</td>
<td>Detailed designs of classes and small clusters of classes are created and implemented. Use of class libraries is an important part of this phase.</td>
</tr>
<tr>
<td>Incremental Integration</td>
<td>The various classes, clusters, and subsystems are integrated incrementally to form larger units. A complete system will evolve after sufficient iterations.</td>
</tr>
<tr>
<td>System Testing</td>
<td>The completed portion of the system is tested from a functional point of view.</td>
</tr>
</tbody>
</table>
9.0 References:


• **Schedule periodic integrations across parallel development teams.** In an incremental approach several pieces are being developed separately, often concurrently. These integrations give more exacting tests of the compatibility of interfaces than can be achieved at the design model level.

• **Use the information gained from testing to guide project management as well as development.** Testing can provide assurance of progress toward the finished product over a series of iterations. As the project progresses, obviously, the number of errors are expected to approach zero. Deviations from the downward trend should be able to be explained by a major restructuring of some piece of the system and this explanation should be part of the testing documentation. Larger projects will usually keep these statistics at the cluster level rather than attempting to manage data on every class. A second interesting metric is the change across iterations in the number of errors being detected. The rate of change in error rates across iterations should also approach zero. As the rate of fault detection goes down, confidence in the product should rise. Table 2 illustrates the matrix approach that can be used to analyze faults and the detection rate.

### 8.0 Summary

A comprehensive testing strategy has been presented that is closely related to the iterative incremental development process model. The approach takes advantage of the products of the development process to construct test cases and conduct tests efficiently. The strategy begins the testing process early in the development cycle so that errors and misconceptions are identified as quickly as possible and corrected for the least cost.

One advantage of this approach is that it improves the probability that project staff will spend the time necessary to achieve quality. The increased overlap between development effort and testing effort provides the time and the motivation to more thoroughly test the components.

One disadvantage of the approach is that it makes the complete testing effort more visible. In some cases this makes it appear that more effort is being directed toward testing than in a typical project. The technique simply makes explicit some processes that have been followed informally and not included in process descriptions.

By integrating the development and testing processes, developers have more reasons to keep documents synchronized with the developed products and the project realizes a greater return on both its development and testing efforts.

### Future Directions

This research is part of a comprehensive effort, under the auspices of the COnsortium for the Management of emerging SOFtware Technologies (COMSOFT) and its corporate sponsors, to provide the necessary infrastructure for supporting object-oriented software development. Plans include the continued development of tools that automate and assist the implementation of the techniques described in this paper as well as new versions of handbooks that elaborate on these testing strategies. Our research group is also actively involved in integrating a metrics program into the testing and development process.

### Acknowledgments:

We would like to thank David A. Sykes and the anonymous CACM reviewers for their constructive comments. We would also like to thank the corporate sponsors and the members of their development staffs who have provided valuable feedback as the process has matured.
7.4 Strategy for Iterative Incremental Testing

Testing products being developed using an iterative incremental approach requires careful coordination and planning. Several issues have been identified and discussed in previous sections. In this section we will summarize these discussions and present a coordinated strategy.

- **Organize the project to provide clear lines of communication for testing information.** This can be accomplished by having public ownership of classes so that it is clear to whom to report problems. Software contracts can also be used to establish clear communication between the provider of a class and the consumers of its services. The incremental nature of our process often separates these two groups into teams that are not even in the same development phase at a point in time. Rapid communication of faults allows for optimum integration of the information into the development process.

- **Assign, and incrementally adjust, development responsibilities to minimize the need for communication.** Class developers should be responsible for the testing and quality of the components they create. Initial object models can be used to determine closely related classes and these can be assigned as clusters to a single team. This sets the stage for early integration and cluster level testing. It is a truism that the architecture of a software system reflects the organization of the development staff. As a result, a successful iterative object-oriented project will need to adjust the team structure as the project progresses and class clusters are reformulated to reflect emerging relationships.

- **Provide developers with the infrastructure needed to support the testing aspect of their task.** Abstract classes for standard exceptions and generic test drivers are examples of the supporting classes that can be defined and provided the developers. An error reporting database, standard reporting formats, and templates for test plans for individual classes can also be provided.

- **Take advantage of the characteristics of the object-oriented products to facilitate the automation of testing.** Classes, with their encapsulation and information hiding properties, provide a continuity from the earliest analysis phases through implementation. We have already discussed the need to test the mappings of a product from one form to another. The object-oriented approach facilitates these mappings since even from the initial models, it is classes that are being mapped, modeled, and implemented. The reuse between classes provided by inheritance can be exploited in the defining of test cases as methods in the test drivers. The project’s development tools that are intended to manipulate classes can be used to manipulate the testing structure as well. Use cases provide a close link between the system test cases and the classes representing the entities referenced in the use cases. These use cases improve the traceability of requirements through the application and into the system test cases. This traceability speeds the communication making it more timely.

- **Establish project-wide guidelines to ensure consistency of products.** These guidelines should include specific goals for adequate testing of the various products. The use of a standard set of algorithms such as those we have presented means that test cases can be understood by other developers. This will facilitate the audit process of selected classes during end of iteration reviews.
The deliverables associated with a class include the class’s specification and implementation, the class’s test suite, and a test driver class. The class will evolve over the iterations and this implies updating of test suites as well as repetition of tests. The developer is in the best position to efficiently manage the evolution of all of these products.

Each development team has responsibility for facilitating the sharing of test cases among developers down a complete inheritance hierarchy. The team also provides an organization within which the first level of integration occurs and the first inter-class interaction test cases are constructed.

The project test team provides a variety of services to the development teams and plays a central role in the testing process. The team establishes and enforces a set of criteria that determine when a developer has performed an adequate number of tests. The team evaluates, and perhaps constructs, testing tools and provides those to the developers. Members of the testing team play an active role in the inspection process and periodically audit the test suites for particularly critical classes.

The project test team also coordinates the interaction of the development teams through the administration of software contracts. The test team assists development teams in writing, negotiating, and modifying the contracts that coordinate the development process.

The system test group provides the traditional system testing responsibilities such as functional testing and stress and performance testing. The group performs additional duties in an object-oriented project. The system group can be assigned the responsibility of validating the use cases against the system requirements. This provides an early validation that the software being built, and based on the use cases, is at least synchronized with the current requirements even though we expect those to change during the life of the project.

The size and complexity of the project will be major factors in determining the appropriate organization. The model described here is appropriate for a moderate to large sized project where a separate test organization is too formal or too much overhead, but the project is too complex to rely on individual developers to handle the coordination. A major benefit of this strategy is that by being an integral part of the project the project test team can more easily coordinate the testing of the various products in the iterative process model. Since members of the team are dedicated to the one project, their knowledge of project details can be deeper and they do not require extra time to remain current.

Smaller projects may not be able to justify all of the levels outlined in this strategy. The informality of a small project and the ease of communication can eliminate the need for software contracts. The developers may still wish to designate a small group of developers that will take the lead in establishing the testing philosophy for all of the developers.

Larger projects or projects with critical, or regulated, quality goals may utilize an Independent Test Group (ITG), a separate organization that is responsible for certifying the quality of the products of the company. The ITG provides, to the development teams, sets of requirements for submission of materials for testing. The ITG may test only the end product of development or every product throughout the life cycle including individual classes. This organizational structure and level of detail is often difficult to accommodate in the iterative approach because this separate organization must be synchronized with the development team’s need for timely feedback.

The major concern that often results in the use of an ITG is the question of how thoroughly individual developers will test their own code. Some believe the objectivity of an independent group is needed to ensure adequate testing. The organization that we have proposed avoids this pitfall by prescribing project wide adequacy criteria, standard test case construction procedures, and a review and audit process.
Within the team communication is informal and quick. Working under one manager, the team usually has a unified set of priorities and a coordinated schedule for delivery.

A **project testing team** is designated to coordinate the testing process. This team, working with project managers, determines a project testing philosophy. The group disseminates the philosophy to the project staff, prescribes procedures to ensure that the philosophy is implemented and provides specific testing tools such as generic test driver classes to the developers. The philosophy includes a set of standard practices such as levels of adequacy for test case coverage.

The **system testing group** offers specialized expertise in testing systems and products and may have special facilities for testing embedded software systems for example. This group can apply corporate standards of quality independent of the pressures of project schedules.

### 7.2 Patterns of Communication

There are several communications patterns that should be established and supported to facilitate the use of testing information in the development process. The coordination of both the testing and development processes should follow the lines of the architecture of the system. This architecture is provided by the inheritance relations among classes, the containment relations between classes and the messaging patterns between objects.

**Developer to developer** - Within a development team there will be informal communication between the developer of a class and a developer who uses that class as part of the definition of another class. This communication will provide feedback about problems encountered when integrating objects from the class into a larger context through containment and when deriving a new class from an existing class. Within a team there should be no need for formal coordination of this communication, but the communication should refer back to the specification of the component. The communication indicates whether there is an incorrect specification or a failure to correctly implement the given specification.

**Developer to team** - A developer will need to provide teammates the test cases developed for a class. This will be necessary if the approach supported by the HIT algorithm is chosen in which test cases are reused down an inheritance hierarchy. This avenue of communication is facilitated by having a standard technique of using a test driver class as the encapsulator of test cases. Then developers subclassing from the original developer’s class will also subclass from that developer’s test driver class.

**Team to team** - As classes are “released” for use by other teams, there must be an avenue for reporting integration problems. This type of communication can be facilitated by the use of software contracts. The contract lists a set of requirements that one team has for the components it will receive from the other team. The contract provides contact people on each team who monitor the contract and respond to requests for modification of the contract or to reports of problems with delivered classes.

### 7.3 A Model Organization for Responsibilities

There are several possible models for assigning the responsibility for coordination of the various products of the testing process. These basic approaches can be mixed and adapted to fit a particular project’s needs. We present here one possible model and discuss some criteria for tailoring the model.

In this model the individual developer performs the class level testing. The developer will want to run numerous small test cases during development anyway so there is not as much extra effort required as would be the case if an independent tester were responsible. The developer constructs components, selects the functional, structural and intra-class interaction test cases, derives a test driver, and executes the test scripts. Each developer owns his/her work and is responsible to the other members of the project staff for meeting the project’s quality standards.
be completed, developers can establish an integration and testing schedule for the level of component represented by the contract. This scheduling becomes particularly important when the two parties in the contract are in separate teams or even in separate development organizations. The advantage of the use of contracts is that they are a standard technique for assuring that appropriate interfaces are built between components independent of the testing issues.

C. Automation - The automation of test case generation, execution, and validation is critical in an iterative environment. Repeatability is also an important feature since each class will evolve across the iterations.

Automation is facilitated by having ready access to the developed products. A number of the techniques described and referenced in this paper derive test cases from specific forms of program representation. For example, a number of functional test cases can be developed from the dynamic model for a class. The algorithm for deriving these test cases, given a simple representation of the state machine, is easy to program. The previously referenced reports on HIT and OATS have sets of algorithms that provide the basis for tool development.

The previously discussed technique of coding the test cases as methods of the test driver also supports automation of the regression testing effort. This approach provides for the inheritance of test cases just as other methods are inherited. This is a natural implementation of the HIT algorithm for reuse of test cases as described in [4]. Changes to classes that are made in later iterations, are reflected in the test drivers and propagated down inheritance structures. The same maintenance advantages touted for object-oriented systems apply to these test driver classes.

D. Use of Development Products - A natural by-product from the interaction of the testing and development processes is the improvement of products from both processes. Table 1 summarizes the set of products from a typical object-oriented development project. It also illustrates the contribution of each product to the generation of test cases. By deriving test cases from these various products, the developers see immediate use for the documents rather than the distant beckoning of the actual implementation phase. Using an almost algorithmic approach to the derivation supports iteration by making it easier to understand how test cases must be refined as the individual products are refined over several iterations.

7.0 Organizational Considerations

The previous discussion about levels of testing activities (e.g., class, cluster, and system) described a set of organizational responsibilities that are not very different from what might be assigned in a procedural approach. The difference is in the amount and the type of coordination that is required. To take advantage of the increased reuse provided by inheritance and the flexibility of the iterative life cycle, an increased amount of coordination is required.

7.1 Groups involved in testing

There are typically four groups involved in the testing process regardless of the organizational strategy: the individual class developers, the development teams, a project testing team, and a corporate level testing group.

The class developer has the most direct knowledge of the class’s specification and is the best source of information on details within the class. The involvement of the developer in testing the individual classes ranges from complete responsibility for class testing to a minimal advisory role in the process.

A development team is a grouping of developers that are working on a related set of classes.
level testing as opposed to testing by some group that is independent of the project structure. The feedback from a group that is separate from the project will most often be too late to be of value.

**B. Coordination** - The approaches used for class, cluster, and system testing will determine how much coordination is required. If the class testing is carried out by the class developer, there will be a decreased need for coordination. If an independent group is responsible for class testing, the group must be informed of the portion of the class that is considered completed and ready for testing and they must, in return, feed information forward into the next iteration as quickly as possible.

These problems occur to differing degrees in the class (i.e. unit) and system testing areas. The classes that compose an object-oriented system are very volatile in the early iterations and some will be eliminated altogether as the design is refined. However, representations of these classes will also be some of the earliest products available for testing. The “system” exists in the early iterations as threads through the object model. Even when classes in the model disappear, these threads are often simply rerouted. This system view, while changing fairly rapidly early in the project, will nevertheless change more slowly than the individual class view. By testing these products, a progression of validated products, each relying on the products from the previous phase, is established.

The incremental approach to development also increases the need for coordination in the testing process. A class is defined in terms of instances of other classes. In order to test a given class, the supporting classes must have been developed at least to the level required by the class in question. This can be accomplished in a number of ways.

**Stubbing** - In this approach, each class is delivered with an accompanying stub class. Appropriate methods in the stub class have been written to return specific dummy values rather than doing an actual computation. This allows the class to be used before the actual implementation has been completed.

A disadvantage of this approach is that stubbing often requires more effort than anticipated. The stub must be capable of responding to multiple messages and it must return the appropriate value each time. This also requires that a separate stub instance be created for each test case so that the stub will produce the appropriate return values for that test case. These multiple instances can be created by overloading constructor methods, but the individual methods must still return the appropriate values. The result of all this is that a stub can evolve into a more complex implementation than the actual method code.

An additional disadvantage is that the work done to create the stub classes does not contribute to the overall completion of the system. With each iteration, the stubs must be reworked to reflect the incremental implementation improvements to the class. This effort can be somewhat reduced using a scheme such as that shown in Figure 6. By deriving the stub class from the base class, the complete and correct implementation for a method can be inherited. Those that are not completed can be overridden to provide the stub behavior.

**Wave front integration** - In our preferred approach, development proceeds across all of the classes at the same time. Developers reach agreement on the functionality to be achieved for each component during each iteration. That functionality should be sufficient for all dependent classes to achieve their negotiated level of behavior. This approach requires much communication, but has the advantage that the work that is done is actual development work and not “extra” work as is the case with the stubbing approach.

The communication required by this approach can be provided by a simple extension to the software contracts described previously for coordinating design. A contract defines the interface between two specific components, classes (illustrated in Figure 8), clusters or subsystems. By adding a section to the contract that specifies dates by which various portions of a component will
ship between Automobile and VehicleTiming.

**Test execution and validation**

Execution of the cases can be controlled by script methods defined in the test driver class. This approach allows for the repetition of tests and for the modification of the testing sequence by providing multiple script methods.

The results of test case execution must be validated via some independent source. This comes in two forms: the creator of the test case and a formal review of the test case database.

In many cases the test case is of the form: \(<\text{initial state, message, resulting state}>\). Both the initial and resulting states are observable states in that there are methods in the class specification that can determine whether the class is in each of the states. The test method can be written to test the validity of the result by comparing the actual state of the object under test with the expected resulting state. The developer is responsible for ensuring that the test case includes the information needed to determine whether the test has been passed or failed. The side effect of returning a value as the result of a method must sometimes also be considered.

The test cases are “tested” during the class development review process. Appropriate individuals, including members of the testing team, are assigned to inspect selected test cases and to validate their correctness. The testing team member assigned to the review will also judge the completeness of the test suite for the class. This will be done by considering the test plan for the class, the coverage criteria mandated by the project’s test plan and the methods used to select the test cases.

One difficulty with validating results is the information hiding property of objects. If there are few accessor methods in the class, it can be very difficult to determine the value of particular attributes. A detailed discussion of this problem is beyond the scope of this paper, but there are several potential solutions to this problem.

1. Test methods can be built directly into the class to overcome this problem but then they must either be shipped as part of the system or they are removed and the product being shipped is different from the product that was tested.
2. For C++ programs, an alternative is to declare the test harness to be a *friend* of the class under test. This allows the harness to have access to the private areas of the class without being a part of the class.
3. The third possibility is the one advocated in this paper. Emphasize functional testing and check for externally observable states for which observer methods are provided. This respects the privacy of the object’s internal representation.

### 6.0 Issues in Integration

Integrating testing and development provides opportunities to improve both processes. These opportunities fall into four categories.

**A. Feedback** - The results of the testing process provide feedback concerning the development products. The feedback is intended to direct the corrective actions of the developer.

Providing feedback is complicated in the context of an iterative life cycle. By the time a product is tested, it may have been modified but not necessarily corrected. Even locating the site of the error may be difficult if an extensive modification has taken place. At the class level this is additional evidence that the class developer should be the one to test the class. In this way, the feedback can be more timely. This also supports the idea of a project test team that is responsible for system
the object under test and can validate the results received back from the messages. These drivers are classes. This approach further integrates the testing and development processes since all of the techniques and tools used to develop production classes may now be applied to tests as well.

The inheritance relation provides an appropriate structure for defining those classes and achieving reuse in the definition process. Individual test drivers are derived from the abstract test driver class or from the test driver of its parent class. The test driver is a management tool that includes methods for:

- logging and displaying test results in a consistent manner across the project,
- methods that represent actual test cases including validation of results, and
- methods that execute a sequence of test case methods.

The test driver class will provide behaviors for logging, displaying and analyzing the test results following the procedures used by a specific company. An abstract class developed for project use, provides the detailed implementations of these output behaviors. Depending upon the project, the output may be a simple stream that creates a file or prints directly to a printer. Alternatively, the fault information may be directed to a database that manages the test reporting process. The test driver creates an instance of the class under test and creates the test environment required such as associated objects that will be sent messages.

Each test case method must set the object under test to the appropriate initial condition, send a sequence of messages, and verify the result of the message sequence. Each test case method should be as independent of other test cases as possible. This can be partially realized by having each method create a “clean” instance of the class under test. This may not always be feasible if creation of the instance requires large amounts of data or other compute-intensive activities. Each test case method should be able to determine whether the object under test has returned the appropriate value and has placed itself in the appropriate state.

The third component in the test driver class is a set of test scripts. These scripts provide a means of sequencing and executing a series of test cases. For most objects, there are particular sequences of methods that are used more often than other sequences. These are not encapsulated in a single convenience method because the object sending the messages must accomplish some computation in between those messages. These scripts allow a class developer to investigate these sequences and to provide appropriate parameters to each of the messages in the sequence.

Figures 6 and 7 illustrate a strategy that coordinates the construction and management of the incomplete production classes and the test driver classes. Figure 6 illustrates the classes and objects that are produced for a specific test case. Vehicle needs an instance of VehicleTiming as part of its definition. The creation of the VehicleTiming instance is always the responsibility of the encapsulating class, Vehicle. Vehicle needs to be able to message an instance of Detector in order to provide some cluster-level behavior. Some external entity will be required to set up a test environment that creates and contains both instances. Automobile is a class derived from Vehicle. It will inherit all of the relationships with VehicleTiming and Detector.

The strategy requires that an inheritance hierarchy of test driver classes be defined in parallel to each inheritance hierarchy of production classes. The specification of the test driver classes is determined by the abstract test driver class. This strategy handles a number of updating issues automatically. In Figure 7, Automobile inherits an association with one class and an aggregation relationship with another from Vehicle. The associated test driver for Automobile inherits how to setup the inherited, associated class. The management of external relationships in which one class is the peer of another class, such as that between Automobile and Detector, is handled by the test driver classes. Classes are responsible for managing their own components, such as the relation-
Production of test cases

Test cases are produced in at least two ways: reusing the test cases from other classes and creating new test cases by examination of the development products.

Functional test cases - These test cases are created by examining the specification of the class. State-based testing [13,14,17] considers the state representation for a class. The state representation is validated by an expert and is compared for consistency with the post-conditions of the class’s modifier methods. The technique selects test cases that exercise each of the transitions in the representation. A breadth-first traversal of the state representation provides the ability to generate test cases that cover single transitions or that group sets of transitions into longer sequences termed “n-way switches” by Chow[2]. The breadth-first sequence ensures that when we test a new transition only previously tested transitions are used.

Alternatively, the test cases may be constructed by examining the post conditions for each method that modifies the state of the object. These conditions provide the boundary conditions for each method. Test cases are constructed to ensure that each method is supporting its post-condition.

This is an example of the use of standard development information (e.g. state diagrams, pre and post-conditions) to drive the testing process.

Structural test cases - The test cases are created by examining the implementation of each method. The significant consideration here is the presence of polymorphic messages that are not bound until execution time.

Interaction test cases - One product of the development process that can be useful during cluster testing is the contract that defines the interactions of the classes involved in a cluster. Software contracts[5] define the interfaces that are provided to other classes in the cluster. The message patterns within a cluster can be exercised by selecting test cases that traverse these interactions.

Management of test cases

The inheritance relation that is used to define the classes can also be used to coordinate and manage the other products of the testing process. Two examples will be considered here.

Interaction test cases - Most of the test cases for a class are developed from information contained within the class. Inter-class test cases, however, require information from two classes and these may well not be owned by the same developer. The project must provide a management strategy that ensures an adequate number of test cases are generated that examine the interactions between a class and its clients.

Certainly many of the interactions between classes are tested incidentally because they are integrated into a single higher level aggregate class. Each time a class defines instances of the two classes within the same environment, the interaction between them is used to provide part of the behavior of the encapsulating class. As test cases for the encapsulating class are defined and executed, their success is some evidence for the correct interaction between the two encapsulated classes. How much of the full range of interaction is really exercised is impossible to answer using this approach. Additionally, the developers of the interacting classes are not aware of the amount of testing being done nor will they necessarily hear about problems that are discovered.

A systematic way to develop these test cases is to require a developer to exercise their class with every class related to their class by a software contract. It is not possible, nor is it even necessary, to test a class in combination with every other class defined in a system. By using the software contract as the guide, the class’s developer is in control of the test cases that are generated and can guarantee that the appropriate coverage is achieved.

Test drivers - Testing each class requires a test driver that can send the appropriate messages to
5.2 Test Case Perspectives
The complete test suite for a class can be conceptually divided into three sections. Each section is constructed based on a different perspective of the component under test. This allows the tester to make statements about the degree to which that perspective has been covered. The three views correspond to the levels of concern of the development team.

Functional - The functional perspective is an external view that considers the behavior promised by the class to those that interact with it. Functional test cases are constructed based on the specification of the component. The specification of a class includes a specification for each method plus a class invariant. Coverage is stated as a percentage of the total class specification that is covered. Typically, every method is executed at least once to verify conformance to the stated pre and post-conditions. Additional test cases may be constructed based on the invariants.

Structural - The structural view is an internal view that is guided by the relationships among individual lines of code. Structural test cases are constructed by identifying individual paths through the code. The coverage will usually be stated in terms of the percentage of the lines of code that have been executed based on the test cases. Typically every line of code is executed at least once; however, it is seldom possible to execute all the paths through the code. A larger percentage of the paths through an individual method can be executed than paths through the complete program.

Functional testing receives more emphasis than structural in object-oriented development. There are several reasons for this. First, the methods in classes are sufficiently small that an aggressive inspection program can catch many statement-level errors early. Second, the information hiding property of objects makes the internals of individual objects less likely to cause problems for other objects. Third, the focus on abstraction, prevalent in object-oriented systems, makes the functional view the natural approach for testing.

Interaction - There are 2 levels of interactions that are of interest to us. Interactions between methods within a class, and interactions between methods in different classes.

At the lowest level, interaction test cases investigate two methods that may directly or indirectly cause each other to produce incorrect results. The test cases are constructed by identifying values that are set or used by two or more methods including cases of parameters being passed between two methods. The two methods may both be in the same class (intra-class) or in different classes (inter-class). Typically each such pair of methods would be executed.

Object technology complicates interaction testing by using dynamic binding. Certain interclass relationships can not be uniquely defined at compile time and thus a definite interaction can not be identified. Strongly-typed object-oriented languages do restrict the associations that can be formed at run-time and make the identification of a set of potential associations possible.

McDaniel and McGregor[12] analyze the possible interactions and utilize orthogonal array algorithms to provide a technique for constructing appropriate test cases. In this approach, the set of legitimate interactions between inheritance hierarchies of classes are identified. For example, in Figure 5, instances of classes in the first hierarchy send a message to an instance of one of the classes in the second hierarchy. The set of interactions can be very large so a sample to be tested is drawn from that set. The sample is drawn using an orthogonal array to select all possible pairs of interacting factors. Pair-wise interactions are first tested, but higher order interactions are not. Higher levels of quality can be systematically attained by increasing the complexity of the interactions that are tested.

5.3 Testing Activities
test results that must be validated.

3. **Use an adequacy criteria that allows one to incrementally increase the level of testing coverage.** A testing method describes a technique that constructs test cases based on some criteria. The method results in a certain amount of coverage of the workproduct being tested. For example, the test cases might be selected to exercise every statement in the code. The testing techniques described here allow for increasing the amount of coverage in a systematic way. Risk analysis is used to identify those classes that should have more testing resources allocated than the norm. McGregor and Dyer[13] describe one such technique for state-based testing. McDaniel and McGregor[12] present an interaction testing technique based on orthogonal arrays that allows the tester to incrementally increase the number of interactions simultaneously being tested.

4. **Use existing workproducts as the basis for deriving test cases.** A critical factor in reducing the amount of perceived effort in the testing process is to use as many products as possible that are created for analysis and design purposes. This reuse of effort supports a more extensive testing process.

5.1 **Levels of Testing**

The testing of code is organized around the recognized units of object-oriented design. Three levels of units are discussed below.

**Class Testing** - This level of testing is somewhat comparable to the unit test of the procedural approach. The name change is not superfluous, however. It is intended to denote a real difference. A class is a different scope from that of a single procedure. With its local attributes and private methods, a class is more complex than a procedure. Class testing is a first level of integration testing. Because of this, “unit testing” in the object-oriented world has a higher payoff that “unit testing” in the procedural world.

It is not productive to attempt to treat anything smaller than a class as an independent test unit. A properly designed class is cohesive and the test driver that would be required to test an individual method would essentially be a reinvention of the class. A class test suite can test those methods that are simple accessor methods before testing the modifier methods. This will reduce the complexity of trying to testing a complete set of methods at one time.

**Cluster Testing** - A cluster of classes is a grouping of cooperating classes. A cluster represents a second level of integration. The focus of cluster testing is the interaction between the instances of the classes in the cluster. It is assumed that each class has been tested individually. To aid in recognizing a cluster, a specification should be developed for each cluster that is similar to the specification for a class. This will include those methods from each class that will be accessed from outside the cluster and pre and post-conditions for each.

Depending upon the range of classes in the cluster and the size of the project, one or more teams will be charged with providing the test suite for a cluster. Special attention must be given to clusters that include classes from more than one team. We will discuss the coordination needed for these products later in the paper.

**System Testing** - The system is tested functionally using test cases derived from the use cases and other system requirements. System testing of object-oriented systems has the same objectives as those for systems developed using other techniques; however, the process used to develop the test cases is somewhat different. There may also be additional objectives related to testing the extensibility of the system. In spite of these differences, there is very little difference in system level functional testing of object-oriented software unless the project has also adopted changes in development objectives (e.g., to emphasize reuse) or adopted a different process model (e.g., an iterative model).
appropriate use cases will be used to verify the completeness and correctness of the formal model. The functional model should be compared to the object model to determine that every class used in an algorithm is represented in the object model and that all of the messages necessary to implement an algorithm are present as associations in the object model.

Figure 3 illustrates the mapping of the use cases into the functional and object models. Every use case should be traceable through the relationships illustrated on the object model. Some of these interactions should be represented as functional models if they involve a significant number of objects within the system. The testing of the mapping is a part of the review process for the system-level object model.

The classes that are in the analysis model are the nucleus of the set of classes in the design model. Therefore the design model will typically already possess the standard qualities of completeness, correctness, and consistency. There are additional qualities that are important in the design model including efficiency, extensibility, testability, and conformance to accepted design practices such as minimal coupling. Each of these qualities is measured by reviewing the designs, comparing to standard practice, and, in some cases, building scenarios to consider “what-if” extensions to the system.

The recent work on patterns[3,7] provides a preliminary basis for testing the correctness of the structures in the design model. By defining “standard” patterns of object specification and interaction, researchers are defining pieces of a system’s architecture and establishing artifacts against which the design can be repeatedly tested across iterations. As designers begin to use the patterns, and to identify them in their design documents, it will be possible to compare the designer’s use of a pattern to accepted practice. For example, Figure 4 shows the well known Model-View-Controller pattern and a system cluster designed around that pattern. The identification of the missing relationship from Window to Document is equivalent to having no communication from the View to the Model. Departures from standard use should not automatically be taken as an error; however, they do indicate areas needing further review. There may be a specific reason for not having the relationship between Window and Document but it is sufficiently different from the typical pattern to be identified as a potential fault in the design.

5.0 Testing Implementations

We previously stated that a basic objective of testing is to uncover the maximum number of weighted errors for a given amount of effort. With the above assumption in mind, we can break this down into the following sub-objectives and heuristics for our implementation testing process.

1. **Create the minimal number of test cases necessary to achieve a given level of test coverage.** The number of test cases that must be created can be reduced if there is a systematic approach to reusing existing test cases. For example, Harrold and McGregor[4] provide a technique, Hierarchical Incremental Testing (HIT) for constructing the minimal number of subclass test cases that are needed to supplement the test cases that are inherited from a parent class.

2. **At any point in the development process, execute the minimal number of test cases necessary to achieve a given adequacy criteria.** In the iterative approach, pieces of a tested design or implementation may change. Should everything be retested when a piece changes? Harrold and McGregor[4] address a related question at the class level. Suppose that a superclass is completely tested. How extensively must a subclass be retested? Some methods will not need to be completely retested after they are inherited, but the test cases are still inherited. Harrold and McGregor also show how to identify those test cases that do not need to be rerun, thus reducing the number of
A description of this set of classes should include a representation of the relationships between the classes, models of the state representation of each class, and models of the important algorithms provided by the set of classes. SASY utilizes the object and dynamic models of the Object Modeling Technique (OMT)[16] and object interaction diagrams[6] to represent the analysis and design information about this set of classes. Any development method will have similar representations and similar documents to represent the static and dynamic behavior of the entities.

Much of the testing of these descriptions will consist of comparisons with the initial representations described above to ensure the accuracy of the translation. However, as the models mature from analysis into the more detailed design models, more information will be available to examine. In particular, with three different views of the system, consistency across the views is an important consideration. The detailed models can be examined for completeness by tracing classes in the initial models into the formal models and for consistency by examining the treatment of individual classes and relationships between classes across the three models.

In a typical development process, the object, dynamic, and functional models are used to represent both the analysis and design models. As the process moves from identifying objects into describing and specifying the objects, these same models can be used and simply elaborated. Different design methods may also add instance diagrams or other supporting detail to provide additional views of the system. There is less likelihood of an error in developing the design model since object-oriented development techniques derive the design models directly from the analysis models.

**Object model** - An object model is created for each set of classes in order to show the classes, their attributes, and the relationships among classes. The object model contains at least all of the classes defined by the CRC cards for that grouping. One test of the object model is to be certain that all of those informally defined classes are present in the formal model. The collaborations from the refined CRC cards should also be accurately represented as various types of associations. To make certain relationships explicit, object models may have circular dependencies among the entities. These paths should be identified and checked for consistency.

Figure 2 illustrates an object model that has been developed from a set of CRC cards. (For those not familiar with the OMT notation, the relationship between Traffic Control Signal and Intersection is aggregation; that between Traffic Control Signal and Crosswalk Signal is specialization; and that between Intersection and Controller is a many to one association.) The testing of this mapping can be part of the object model review. Each component of the previous model should be accounted for in the new model. Elements may be eliminated, split, or combined but the materials supporting the new model should describe what has happened to each element. The mapping should consistently handle elements. That is, similar situations should be handled the same by the mapping. Testing at this level is a review of the two models side-by-side to determine that all elements from the first model are appropriately represented in the second.

**Dynamic model** - This type of model is developed at a variety of levels, whenever there is a need to represent the dynamic behavior of a class or set of classes. A dynamic model includes a set of states and the transitions between those states. The states should be observable, and therefore confirmable. The methods through which these states are observed and by which the state is changed are behaviors of the class. The appropriate object model should be checked to determine that those behaviors are present. The dynamic model should be verified against the functional model to verify that the sequence of messages required for an algorithm lead to valid sets of state transitions for the participating classes.

**Functional model** - The functional models will include representations of the algorithms that are significant to the set of classes. Each model will include the sequences of messages corresponding to some algorithm. In cases where the algorithm corresponds to system functionality, the appro-
fourth, perhaps these three responsibilities should be combined. The creator of Detector might find that many of the requests to read status are immediately followed by a request to reset itself. Were this true, a single responsibility, ReadandClear, would be added as a responsibility.

• The resulting set of cards is reviewed by domain experts to judge the model.

By applying these simple checks, problems can be quickly found and the feedback can lead to an improved set of CRC cards. Since the CRC cards will be used to produce the object model, finding mistakes at this point makes the creation of that model more efficient and increases its quality. The formal models of classes are developed as component specifications. Each method has an associated set of pre and post-conditions and the class as a whole supports a class invariant. The class specification is evaluated by a comparison against its CRC description and against its role in the total system. The class model is complete if the specification includes methods that implement all of the required responsibilities. It will be judged correct at a specification level if a prototype implementation illustrates that the specification provides the appropriate behaviors. The implementation will be judged correct only after an actual implementation is tested as described in a later section.

4.4 System Models

The analysis models that are developed in a project are either domain-related or application-related. Each domain-related model is intended to represent some area of knowledge whose scope is roughly determined by the depth and breadth of our interest. A typical application will involve multiple domains and therefore multiple models. Testing these models will require the collaboration of persons knowledgeable about each domain. The models are compared to standard knowledge in the field as found in official standards and the literature of the field.

The application-related models provide early views of the actual system being built. The models of a set of relevant domains are used to develop the model for an application. The application model adds the system-specific details needed for domain entities to be related and productive in a computer system. The collaboration of potential users of the system are needed for this testing. The model is tested from the perspective of the system use cases.

The application model is developed by combining several domain models, the user interaction model, and additional classes that support the application. The mapping from these various models must be complete and correct. The model is not complete in the sense of including everything from the initial models. It should provide those entities that are necessary to model the portion of the domains required by the application. The application model should be compared to the set of use cases to ensure completeness. The review team should be able to trace every user interaction into the set of classes that define the interface of the application. The CRC cards should provide a basis for verifying that each class is correctly mapped. The application model should contain correct descriptions of the classes that are mapped into it and it should account for the relationships among those classes. With models being combined, consistency is a concern. Domains often overlap and the new model needs to be reviewed to determine whether similar entities from different domains have been recognized, integrated or differentiated, and handled consistently across the model. The resulting application model should combine the various views of the system into a cohesive whole. The entities from the various models should interact and these interactions should have been captured in the original models as interface issues.

The early modeling techniques, such as the CRC model, have the flexibility that supports quick and easy modifications. As those models become more stable there is a need to support a more precise and integrated view of the classes. These models describe a set of classes that may represent a domain of knowledge, a complete application, or a subsystem of some larger product. A de-
a set of use cases that can be used to guide system development without repeated reference to the much more detailed requirements.

The model is tested for consistency by examining the hierarchical structure of the use cases. Similar interactions from the specifications should all map into a single hierarchy. A similar exercise can be carried out for scenarios by grouping them based on common entities required in the model. Again similar interactions should map into a single grouping.

The more formal specification model is often tested by building an executable model. The development of these prototypes provides two tests of the models. First, during the development of the prototype, many inconsistencies will be identified and that feedback should directly affect the models. The second test, the execution of the model, allows developers to conduct “what-if” tests to determine the completeness and correctness of the representation. In these tests the developer uses the executable model in the same ways that users will use the completed system, as described in the use case model. This examination will identify missing relations among classes as well as finding some relations in the model that should not be present.

4.3 Individual Class Models

The initial models for individual classes are often Class - Responsibility - Collaborator (CRC) cards[1]. This very simple notation links each class to its set of collaborators, classes that provide service to the originating class. The usefulness of the CRC model is in its informality and ease of modification. The major errors that can be made in the CRC model include naming as a collaborator a class that does not exist; not including concepts that should be part of the analysis model; a collaborator not providing a needed service as one of its responsibilities; and grouping together unrelated responsibilities.

A testing-by-inspection process can be built into the CRC construction exercise by providing a checklist for developers to use during the exercise. The checklist considers the interconnections illustrated in Figure 1. The individual creating a CRC card is responsible for using the checklist to test the class they are describing. The steps and checks are as follows:

- Locate the CRC card for each of the listed collaborators. The creator of Controller will find the creator of Detector.
- Inspect each collaborator’s CRC card to determine if the delegated responsibility is included on the collaborator’s card. The creator of Controller asks if a Detector will acknowledge the presence of Vehicles when queried.
- Record the classes that come to you seeking service. Note the services that are being requested by these classes. The creator of Controller maintains a list of all cards listing Controller as a collaborator, such as an Intersection (not shown).
- Use this tracking information to determine whether there are clear groupings of responsibilities/classes that are disjoint or largely so from other groupings. This points to the need for a realignment, creating classes for each of the groups of responsibilities from the one existing class. The creator of Controller might find that responsibilities for controlling the lights at a single intersection was a different set of responsibilities than coordinating the settings of signals between two adjacent intersections. Two classes such as LocalController and NetworkController might be created.
- The tracking information can also lead to cleaner definitions of the responsibilities. For example, if every class that uses the fifth responsibility of a class also uses the second and
answer many questions about the content of the model.
The testing carried out during reviews can be integrated with the development reviews that are
typically conducted at the end of each iteration. The primary difference between the two reviews
is the level of specificity. The development review usually includes a presentation of the models
developed during the iteration. The testing portion of the review includes the use of test cases to
exercise the model at a detailed level.

The development of model-level test cases provides a vehicle for making these tests repeatable, a
necessity in an iterative environment. Each test case is a detailed scenario that requires some ac-
tion on the model such as instantiating a portion of the model or building extensions at specific lo-
cations in the model. These cases should be reviewed by the domain experts and applied during
the design phase. The nature of these test cases usually requires explicit verification by experts
since correct results are difficult to formally specify.

The test cases should address the goals of those models. The goals of object-oriented system de-
sign usually include being reusable, extensible, and reconfigurable. Every model's test suite
should include test cases that ascertain whether the model fully represents what is currently in-
tended. Specific scenarios are created, driven through the model using instance diagrams that
show the resulting network of objects, and this network is examined to determine that the repre-
sentation meets the criteria stated above. For those models that are explicitly intended for reuse,
such as that for a framework, some cases that address hypothetical extensions should be included
in the test suite. These test cases require the specification of proposed new classes that extend the
existing design. Instance diagrams can then be created to examine the viability of the extension.
The key element in both cases is that explicit test cases be constructed so that specific objects can
be created from the representation.

4.2 Specification Models
A major problem in software development is errors in the specification. In spite of the high fre-
quence of errors in specifications and the costly consequences of these errors, there exist few
 techniques for testing specifications. Although object-oriented techniques do not eliminate this
problem, we have found that user participation in the testing of domain models coupled with de-
laying application analysis until after domain analysis results in much higher quality specifica-
tions.

Errors, ambiguities, and omissions not apparent in the specification often become more obvious
when use cases[6], descriptions of specific interactions between the system and a specific user, or
scenarios[18], informal descriptions of how the system will be used, are developed as part of the
work products. This is specially true when the use cases, or scenarios, explicitly include a “ratio-
nale” section. When customers review the developers' understanding of their requirements they
are better able to pinpoint omissions, errors and misleading information in the specification.

The testing of the use case model's content is conducted by a review group that should include
representatives from the testing team, members of the development organization and potential us-
ers of the system. A testing-by-comparison approach is carried out by comparing the interaction
model to some version of the system requirements. This activity is completely integrated into the
customer review cycle.

The interaction model is correct if the user interactions portrayed in the model accurately describe
valid user interactions. In part this can be determined from the requirements. This review also has
the effect of testing the mapping of the use cases from the requirements.

The model is complete if all legal user interactions with the system are represented. This includes
identifying all users of the system as well as all of their possible uses of the system. This results in
al. consider three aspects of the model: syntax, semantics and pragmatics. They identify four criteria for quality models: correct, complete, valid, and comprehension.

We will utilize a portion of their framework to address our more narrowly focused concerns with fault detection. We consider both syntactic and semantic correctness and completeness as outlined in their framework. We use consistency, which is included within their definition of validity, but the other facets of their definition of validity pertain more to a quality perspective than fault detection.

These attributes must be interpreted in the context of the iterative incremental approach. That is, early versions will not be expected to attain the same level of detail as later versions. The emphasis on the testing process is the feedback into the development process.

Correctness and completeness are judged against the aspects of reality which the model is intended to represent. The set of analysis entities are semantically correct if they are an accurate model of reality. Early models will not contain the level of detail that later versions will, but the information that is present should be judged to be accurate at that level of abstraction. Correctness criteria can be applied to the syntax of the model as well.

The model is complete if it is judged that the entities describe the aspects of the knowledge being modeled in sufficient detail for the goals of the current iteration. In particular, the model should be understandable to domain experts as a comprehensive representation of standard knowledge within its field. For software engineers who are not domain experts, the model should contain sufficient information to support their development efforts during the current iteration.

Consistency is judged by considering the relationships among the entities in the model. An inconsistent model has representations in one part of the model that are not correctly reflected in other portions of the model. These may be contradictions or differences in level of detail. Incremental models will be segmented across the increments, but the interfaces between the pieces of models should contain sufficient “interface” entities to support a consistent treatment of entities across model boundaries.

The process of mapping one model into another model can introduce errors whenever one model is used to build another model. Testing of the new model is partially accomplished by comparing it to the original model. The transformations from one representation to another are usually made to support a more precise notation. For example, the initial exercises for identifying objects need more flexibility than the notation that represents the formal specifications of those entities.

One of the ways in which object-oriented methods result in higher quality code is by having fewer mappings and transformations that are less radical than in other development methods. Analysis models, design models, and implementation models all use classes and instances of classes as the primary vehicles of representation. Even though a model shows some particular view that is different from another model, all of the models are illustrating aspects of classes.

The correctness of a mapping is judged by comparing the original model to the derived model to determine that the information has been accurately represented. The mapping should be complete in that every entity in the original model should be accounted for in the transformation. The mapping may result in an existing entity being eliminated from future models or that entity may be split into two or more smaller more specific entities. The mapping should maintain the same relationships between entities in the new model as existed between the entities in the previous model.

Our testing process uses a number of techniques to apply these criteria to the various workproducts. Reviews of the products are carried out by the model developers assisted by experts in the domains in which the systems work, experts in the object-oriented development process and potential users of the system. Products may also be compared to previous versions of the same product or to earlier products. Some of the models will be executable and their execution will help
customers and on the amount of rework required to fix them. We will use risk analysis to identify those classes that are more “critical” to the system. The definition of critical will vary from one project to another and from one company to another. Critical classes will be more closely examined and consume a disproportionate share of the testing resources.

Three major categories of testing activities are listed below. The goal of the first is to locate faults in the software development process itself. The goal of the second is to locate faults in non-executable products such as models, CRC cards, designs and frameworks. The goal of the last is to locate faults in the executable code.

The 3 categories are:

1. **Testing of the development process and the supporting documents** - There is a need to evaluate and improve the processes as much as the products. The process is embodied as a set of activities and supported by a set of documents that describe the activities. The iterative approach supports continuous improvement through feedback. Specifically, the testing of the process examines the goals for each iteration and evaluates the level of achievement at the end of the iteration. It also examines a cross section of successive iterations to identify problems and improvements in a particular activity across the iterations. Other factors to test include the continuity between successive phases in the life cycle.

2. **Testing analysis and design models** - The object-oriented approach places increased importance on the analysis and design models and the identification of abstractions. Since these directly shape the code, testing analysis models and designs is especially important. As with the testing of the process, comparison between versions of products across iterations is one approach for testing as is the examination of the mapping of the products from one stage for the next.

3. **Static and dynamic testing of implementations** - Our goal is to locate as many errors as possible early in the process, but errors in the code of large complex systems are inevitable. Many procedural projects are relying more heavily on code inspections (static testing) to find simple logic errors.[15] The smaller methods used in object-oriented systems are even more amenable to this type of detection. Dynamic testing concentrates on executing test data to find faults in the code.

In this paper we will focus on the last 2 categories of testing activities. We believe that reviews of the development process are an important and very high leverage part of testing, but the topic is beyond the scope of this paper.

### 4.0 Testing Models

The object-oriented approach emphasizes the creation of models of analysis and design information. When an iterative process model is adopted, these models begin as informal representations and evolve into more formal, and more precise, models. The testing of the informal models must be fast and cheap since the models change rapidly. The feedback from this testing provides direction to the evolution of the models by pointing out missing entities or inconsistencies in the treatment of entities. The formal models are tested more carefully and completely since they will have a longer life and play a central role in the shape of the final system.

### 4.1 Criteria for Testing Models

We will consider three interrelated attributes that are common to all models: correctness, completeness, and consistency. These attributes are similar to those presented in Lindland et al.[10] who provide a framework for achieving quality in conceptual models. In particular, Lindland et
• the preconditions of each method in a child class must be no stronger than the preconditions of that method in the parent class;

• the postconditions of each method in a child class must be no weaker than the postconditions of that method in the parent class; and

• the invariant of a child class is a superset of the invariant of its parent class(es).

This approach to the use of inheritance supports the development of hierarchies that represent classifications of the concepts contained in the structure. Each class represents a special case of its parent. This produces a logical structure in which once a method is part of the specification of a superclass, it is never removed and its intention is never changed in any subsequent subclass. This results in a semantically equivalent set of classes that may be safely polymorphically substituted for one another.

3.0 An Integrated Development/Testing Process

The integration of the development and testing processes can best be described by paraphrasing a description of iteration, often attributed to Grady Booch, “analyze a little, design a little, code a little, test a little” to read “analyze a little, test a little, design a little, test a little, code a little, test a little”. In other words, testing is continuously interwoven into the development process. This not only locates faults early, it makes succeeding phases less likely to create new faults based on existing ones. This assumes an iterative incremental approach in which the steps of the development process are repeated and the product is developed through successive refinements.

The goals of a testing process are twofold:

• A testing process can give some level of confidence that a software system meets, under specific conditions, at least certain of its objectives.

• A testing process can detect faults in the software.

From the earliest iterations through the software development activities, we are interested in both goals. Before proceeding to a subsequent iteration we would like to have some level of confidence that the work performed to date is correct, but at the same time we know that in large complex projects errors do occur and we want to get them out at the earliest possible time so as to minimize the costs of rework and keep quality high. Finding faults early not only keeps costs down but increases quality because systems reworked and debugged at the last minute often contain workarounds and other compromises to the integrity of the design. Furthermore these last minute changes are often not reflected in the analysis and design documents.

To allow for early fault identification, the viewpoint presented here is that “testing” goes beyond the execution of code. It is used here to indicate activities such as formal reviews, comparisons, or any activity in which criteria are used to detect misconceptions, omissions, and mistakes and to provide feedback for improvement. In fact our definition of testing will include all activities in which our intent is to detect faults. This view of testing supports the increased emphasis on “front-end” activities prevalent in object-oriented methods.

A basic objective of testing is to uncover the maximum number of weighted errors for a given amount of effort. We use the term weighted errors since some errors have a larger impact, both on
2.0 Background

In writing this paper, we have assumed that the reader is familiar with the basics of object-oriented analysis, design and programming. For those who are not, Korson and McGregor[8] provide an overview of the basic concepts and issues. We also make assumptions about the process model adopted by a project and the development method used during the project.

2.1 Basic Shape of the Process Model

An iterative approach to development is often used in projects that adopt an object-oriented perspective. The information hiding and loose coupling attributes of object technology facilitate the rework that results from the iterations. Most of the widely-used object-oriented methods, described in the literature, advocate the iterative approach. Thus a realistic testing process for object-oriented projects should accommodate this style of development.

The Software Architects’ SYnthesis Model (SASY)[9] is a process model that is a synthesis of the most effective aspects of a number of object-oriented methods. These include (1) domain analysis as a source of classes and models; (2) an iterative structure that supports successive refinement of workproducts; and (3) an incremental approach that solves large problems a piece at a time. The side-bar shows a listing of the key activities as well as the flow of the iterative cycle of development phases. We will use this model as we describe the interaction between the testing and development processes.

2.2 Development Process Products

The development process produces many workproducts. Those of interest to testers include models, designs, and implementations. Each of these products should be tested as a product independent of any other, but there are also many inter-relationships in which one product contributes to the development of test cases for a related product. The sections of the paper on testing models and implementations will detail techniques that can be used with each of the development products.

The relationship of the development process to the various development products is often a source of confusion to those new to the object-oriented paradigm because they may be used to having unique activities and notations for each phase of the development process. For example, one might be accustomed to using data flow diagrams to represent analysis information and structure charts to represent design information. In the object-oriented process, workproducts flow across process phase boundaries. For example, object models that are started during Domain Analysis are focused and refined during Application Analysis, and clustered and enhanced with design mechanisms during Application Design. From a testing point of view this implies that many of the workproducts will have to be iteratively and incrementally tested.

In the side-bar on SASY products we have listed the workproducts that we will reference in the rest of the paper. We recognize that there are many variants on the development process and workproducts produced, but it should be relatively easy to adapt our recommendations to any process for object-oriented software development.

2.3 Inheritance Model

A central issue to the development effort for any object-oriented project is the philosophy concerning the use of inheritance. Many of the algorithms and techniques described in the testing process assume the use of a “strict” inheritance approach. This approach is similar to several other design guidelines including Liskov’s [11]. In this philosophy of use of inheritance:
Integrating Object-Oriented Testing and Development Processes

John D. McGregor
Dept. of Computer Science
Clemson University
Clemson, SC 29634-1906

Timothy D. Korson
COMSOFT
P. O. Box 263
Clemson, SC 29633

1.0 Introduction

Testing is one of the areas of software engineering in which the gap between research knowledge and actual practice is very large. One of the reasons often cited is the amount of “extra” effort that is required in practice to develop the infrastructure necessary for a comprehensive testing process. A primary reason that the effort is considered extra is that the testing process is often not integrated with the development process.

The integration of the testing and development processes provides an opportunity for better coordination between the deliverables of the two processes. This coordination can lead to less redundancy between the processes and thus to a reduction in the overhead in both processes.

In this paper, we will present an integrated development/testing process model that utilizes an iterative approach to apply object-oriented methods to software system development. We will consider several problems encountered in the testing portion of this process and discuss possible solutions to those problems. We will take the point of view that “testing” includes the verification and validation of any product and not just the execution of code. While we will briefly outline a development method, our focus will be on the testing portion of the integrated process.

In the next section, we give an introduction to the process model and other assumptions necessary for the testing and development processes. We then present a comprehensive testing process and discuss the techniques for testing specific work products. Finally we describe issues in the integration of the testing and development processes and discuss possible approaches for organizing the testing effort.

1. This research was partially supported by a grant from COMSOFT, IBM, and BNR.