|  |  |  |
| --- | --- | --- |
| http://www.stickyminds.com/images/c.gif | | |
| Article Picture | http://www.stickyminds.com/images/c.gif | http://www.stickyminds.com/images/c.gif **Use Cases and Testing** **Testing UML Models** http://www.stickyminds.com/images/c.gifBy Lee Copeland http://www.stickyminds.com/images/c.gif | http://www.stickyminds.com/images/c.gif |
| http://www.stickyminds.com/images/c.gif | | |  |

http://www.stickyminds.com/images/c.gifIn today's testing world there is good news and bad news. The good news is that, more and more, testers are being asked to evaluate the quality of object-oriented analysis and design work earlier in the development process. The bad news is that most of us do not have an extensive background in the object-oriented paradigm or in UML (Unified Modeling Language), the notation used to document object-oriented systems.

This is the first in a series four of articles written to

* introduce you to the most important diagrams used in object-oriented development (use case diagrams, sequence diagrams, class diagrams, and state-transition diagrams)
* describe the UML notation used for these diagrams
* give you as a tester a set of practical questions you can ask to evaluate the quality of these object-oriented diagrams

We will use three independent approaches to test our diagrams:

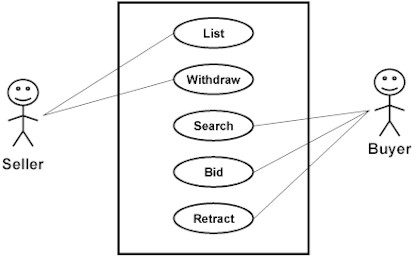
* syntax   
  "Does the diagram follow the rules?"
* domain expert   
  "Is the diagram correct?" "What else is there that is not described in this diagram?"
* traceability   
  "Does everything in this diagram trace back correctly and completely to its predecessor?" "Is everything in the predecessor reflected completely and correctly in this diagram?"

For this set of articles we will use a case study: a Web-based, online auction system that I invented: F-LAKE. Yes, I invented the idea of online auctions. I'm not sure why F-LAKE never caught on.

**Use Cases**  
A use case is a scenario that describes the use of a system by an actor to accomplish a specific goal. What does this mean? An actor is a user playing a role with respect to the system. Actors are generally people although other computer systems may be actors. A scenario is a sequence of steps that describe the interactions between an actor and the system. The use case model consists of the collection of all actors and all use cases. Use cases help us

* capture the system's functional requirements from the users' perspective
* actively involve users in the requirements-gathering process
* provide the basis for identifying major classes and their relationships
* serve as the foundation for developing system test cases

The UML notation for use cases is



The rectangle represents the system boundary, the stick figures represent the actors (users of the system), and the ovals represent the individual use cases. Unfortunately, this notation tells us very little about the actual functionality of the system.

Alistair Cockburn has proposed the following format for defining the details of each use case, which I have used on a number of projects. I've included an example use case for F-LAKE: "Bid On An Item."

|  |  |  |
| --- | --- | --- |
| **Use Case Name** | Bid on an Item | |
| **Context Of Use** | Bidder wants to purchase an item | |
| **Scope** | System | |
| **Level** | Primary task | |
| **Primary Actor** | Bidder | |
| **Stakeholders and Interests** | Seller, FLAKE.com | |
| **Preconditions** | None | |
| **Success End Conditions** | Bid has been processed against the item | |
| **Failed End Conditions** | Bid has not been processed; item and its current bid price remain unchanged | |
| **Trigger** | Bidder enters a bid into FLAKE | |
| **Description** | **Step** | **Action** |
|  | 1 | Bidder enters a "Bid on Item" request |
|  | 2 | FLAKE bids up the currentBid by looping through all active bidders on this item |
|  | 3 | FLAKE waits for BidTimer to expire |
|  | 4 | FLAKE notifies winning and losing bidders |
| **Extensions** | None | |
| **Variations** | None | |
| **Priority** | Critical | |
| **Response Time** | Within 5 seconds of entry | |
| **Frequency** | Approximately 100,000 times per day | |
| **Channels To Primary Actor** | Interactive through a Web interface | |
| **Secondary Actors** | None | |
| **Channels To Secondary Actors** | N/A | |
| **Due Date** | 1 Feb 2000 | |
| **Open Issues** | None | |

**Syntax Testing**Let's begin with the simplest kind of testing—syntax testing. When performing syntax testing, we are verifying that the use case description contains correct and proper information. We ask three kinds of questions: Is it complete? Is it correct? Is it consistent? In one project I worked on, more than half the use cases failed syntax testing. Should we proceed with further implementation and testing given that level of quality?

Oh, I almost forgot. And you *must* keep this a secret. You do not need to know the answers to any of these questions before asking them. It is the process of asking and answering that is most important. Listen to the answers you are given. Are people confident about their answers? Can they explain them rationally? Or do they hem and haw and fidget in their chairs or look out the window or become defensive when you ask? Now for the questions:

Complete:

1. Are all use case definition fields filled in? Do we really know what the words mean?
2. Are all of the steps required to implement the use case included?
3. Are all of the ways that things could go right identified and handled properly? Have all combinations been considered?
4. Are all of the ways that things could go wrong identified and handled properly? Have all combinations been considered?

Correct:

1. Is the use case name the primary actor's goal expressed as an active verb phrase?
2. Is the use case described at the appropriate black box/white box level?
3. Are the preconditions mandatory? Can they be guaranteed by the system?
4. Does the failed end condition protect the interests of all the stakeholders?
5. Does the success end condition satisfy the interests of all the stakeholders?
6. Does the main success scenario run from the trigger to the delivery of the success end condition?
7. Is the sequence of action steps correct?
8. Is each step stated in the present tense with an active verb as a goal that moves the process forward?
9. Is it clear where and why alternate scenarios depart from the main scenario?
10. Are design decisions (GUI, Database, …) omitted from the use case?
11. Are the use case "generalization," "include," and "extend" relationships used to their fullest extent but used correctly?

Consistent:

1. Can the system actually deliver the specified goals?

**Domain Expert Testing**After checking the syntax of the use cases we proceed to the second type of testing— domain expert testing. Here we have two options: find a domain expert or attempt to become one. (The second approach is always more difficult than the first, and the first can be very hard.) Again, we ask three kinds of questions: Is it complete? Is it correct? Is it consistent?

Complete:

1. Are all actors identified? Can you identify a specific person who will play the role of each actor?
2. Is this everything that needs to be developed?
3. Are all external system trigger conditions handled?
4. Have all the words that suggest incompleteness ("some," "etc."…) been removed?

Correct:

1. Is this what you really want? Is this all you really want? Is this more than you really want?

Consistent:

1. When we build this system according to these use cases, will you be able to determine that we have succeeded?
2. Can the system described actually be built?

**Traceability Testing**Finally, after having our domain expert scour the use cases, we proceed to the third type of testing—traceability testing. We want to make certain that we can trace from the functional requirements to the use cases and from the use cases back to the requirements. Again, we turn to our three kinds of questions: Is it complete? Is it correct? Is it consistent?

Complete:

1. Do the use cases form a story that unfolds from highest to lowest levels?
2. Is there a context-setting, highest-level use case at the outermost design scope for each primary actor?

Correct:

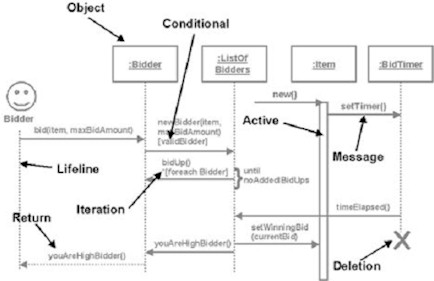
1. Are all the system's functional requirements reflected in the use cases?
2. Are all the information sources listed?

Consistent:

1. Do the use cases define all the functionality within the scope of the system and nothing outside the scope?
2. Can we trace each use case back to its requirement(s)?
3. Can we trace each use case forward to its class, sequence, and state-transition diagrams? http://www.stickyminds.com/images/c.gif  
   http://www.stickyminds.com/images/c.gif

**Sequence Diagram**A sequence diagram describes how groups of objects collaborate in accomplishing some system behavior. This collaboration is implemented as a series of messages between objects. Typically, a sequence diagram describes the detailed implementation of a single use case (or one variation of a single use case). Sequence diagrams are not useful for showing the behavior within an object. Consider using state-transition diagrams for that purpose.

The UML notation for sequence diagrams is shown below:



**Notation**For those not familiar with the notation used for sequence diagrams, some explanation is in order.

*Object.* Each of the objects that participate in the processing represented in the sequence diagram is drawn across the top. Note that objects are used in this diagram while classes are used in use cases, class diagrams, and state-transition diagrams.

*Lifeline.* A dotted line is dropped from each object in the sequence diagram. Arrows terminating on the lifeline indicate messages (commands) sent to the object. Arrows originating on the lifeline indicate messages sent from this object to another object. Time flows from top to bottom on a sequence diagram.

*Active.* To indicate that an object is executing, i.e., it has control of the CPU, the lifeline is drawn as a thin rectangle.

*Message.* A horizontal arrow represents a message (command) sent from one object to another. Note that parameters can be passed as part of the message and can (optionally) be noted on the diagram.

*Return.* When one object commands another, a value is often returned. This may be a value computed by the object as a result of the command or a return code indicating whether the object completed processing the command successfully. These returned values are generally not indicated on a sequence diagram; they are simply assumed. In some instances the object may not be able to return this information immediately. In this case, the return of this information is noted on the diagram later using a dotted arrow. This indicates the flow of information was based on a previous request.

*Conditional.* Square brackets are used to indicate a conditional, i.e., a Boolean expression that evaluates to TRUE or FALSE. The message is sent only if the expression is TRUE.

*Iteration.* Square brackets preceded by an asterisk (\*) indicate iteration. The message is sent multiple times. The expression within the brackets describes the iteration rule.

*Deletion.* An X is used to indicate the termination (deletion) of an object.

**Syntax Testing**Let's begin with the simplest kind of testing—syntax testing. When performing syntax testing, we are verifying that the sequence diagram contains correct and proper information. We ask three kinds of questions: Is it complete? Is it correct? Is it consistent?

Do you remember our secret from the last article? You do not need to know the answers to any of these questions before asking them. It is the process of asking and answering that is most important. Listen to the answers you are given. Are people confident about their answers? Can they explain them rationally? Or do they hem and haw and fidget in their chairs or look out the window or become defensive when you ask? Now for the questions:

Complete:

1. Does each object required for the interaction appear on the diagram?

Correct:

1. Have all objects not required in the interaction been removed from the diagram?
2. Does each object’s lifeline begin and end at the proper time?
3. Is each object’s activation described properly?
4. If the object’s lifetime terminates, is it indicated with an X?
5. Is each message well named with a strong verb?
6. Are proper parameters included for each message?
7. Are conditional branches drawn properly?

Consistent:

1. Do conditionals cover all of the cases?
2. Have any overlaps of conditionals been removed?

**Domain Expert Testing**After checking the syntax of the sequence diagrams, we proceed to the second type of testing—domain expert testing. Again, we have two options: find a domain expert or attempt to become one. (The second approach is always more difficult than the first, and the first can be very hard.) Continuing, we ask two kinds of questions: Is it complete? Is it correct?

Complete:

1. Are all the ways that things could go right identified and handled properly?
2. Are all the ways that things could go wrong identified and handled properly?
3. Does the main success scenario run from the trigger to the delivery of the success end condition?

Correct:

1. Does the sequence diagram show each step that must be executed to implement the function?
2. Can each step actually be implemented?

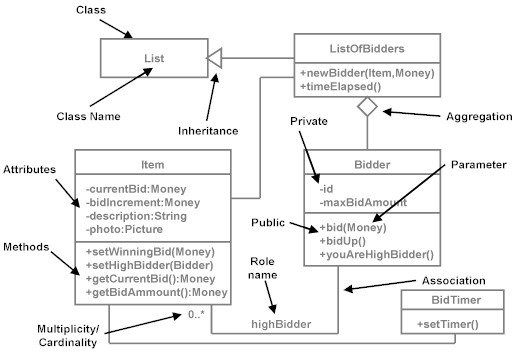
**Traceability Testing**Finally, after having our domain expert scour the sequence diagrams, we proceed to the third type of testing—traceability testing. We want to make certain that we can trace from the use cases to the sequence diagrams and from the sequence diagrams back to the use cases. Again, we turn to one question: Is it consistent?

Consistent:

1. Is each use case represented by at least one sequence diagram?
2. Does each actor appear on at least one sequence diagram?

**Class Diagrams**   
A class diagram describes the classes that make up a system and the static relationships between them. Classes are defined in terms of their name, attributes (or data), and behaviors (or methods). The static relationships are association, aggregation, and inheritance.

The UML notation for class diagrams is shown below:



**Notation**   
For those not familiar with the notation used for class diagrams, some explanation is in order.

***Object***. A specific entity or concept that has meaning in an application domain.

***Class***. A definition of a set of potential objects that have the same data, behavior, and relationships.

***Attribute***. A data value defined in a class and held within an object that has meaning within the application domain.

***Behavior***. A service defined in a class and provided by an object.

***Method***. The implementation of a behavior in an object-oriented programming language.

***Association***. A "peer-to-peer" relationship between classes.

***Aggregation***. A "whole/part" relationship between classes.

***Inheritance***. A "generalization/specialization" relationship between classes.

***Cardinality/Multiplicity***. The minimum and maximum number of objects that participate in an association or aggregation. The common (interesting) ones are 0..\*, 0..1, 1..\*, and 1..1

***Polymorphism***. The ability to send a message to an object without knowing its specific class.

**Syntax Testing**   
Let's begin with the simplest kind of testing—syntax testing. When performing syntax testing, we are verifying that the class diagram contains correct and proper information. We ask three kinds of questions: Is it complete? Is it correct? Is it consistent?

Do you remember our secret from the previous articles? You do not need to know the answers to any of these questions before asking them. It is the process of asking and answering that is most important. Listen to the answers you are given. Are people confident about their answers? Can they explain them rationally? Or do they hem and haw and fidget in their chairs or look out the window or become defensive when you ask? Now for the questions:

Complete:

1. Does each class define attributes, methods, relationships, and cardinality?
2. Is each association well named?
3. Is each association’s and aggregation’s cardinality correct?

Correct:

1. Are all attributes private?
2. Are all parameters explicit rather than being embedded in method names?
3. Do all subclasses implement the "is-a-kind-of" relationship properly?
4. Are all object states represented explicitly using states and transitions rather than as subclasses?
5. In inheritance structures, are all attributes and methods pushed as high in the inheritance structure as is proper?
6. Are all polymorphic methods within related subclasses identically named?
7. Does each association reflect a relationship that exists over the lives of the related objects?

Consistent:

1. Are each 0..\* and 1..\* relationships implemented with containers/collectors?
2. Are each association’s cardinalities consistent (instantaneous vs. over-time)?

**Domain Expert Testing**   
After checking the syntax of the class diagrams, we proceed to the second type of testing—domain expert testing. Again, we have two options: find a domain expert or attempt to become one. (The second approach is always more difficult than the first, and the first can be very hard.) Continuing, we ask two kinds of questions: Is it correct? Is it consistent?

Correct:

1. Is each class named with a strong noun?
2. Have all redundant, irrelevant, or vague classes been removed from the diagram?
3. Is each attribute defined within the proper class? Is it of the correct type?
4. Is the visibility of each attribute correct?
5. Are the default values of each attribute specified correctly?
6. Is each attribute essential rather than computable from others?
7. Is each method in the correct class?
8. Are all method names strong verbs?
9. Does each method take the correct input parameters and return the correct output parameter?
10. Is the visibility of each method correct?
11. Does each method implement one and only one behavior?
12. Is the public interface free from unnecessary methods?

Consistent:

1. Is the class diagram drawn at the appropriate level: conceptual, specification, or implementation?

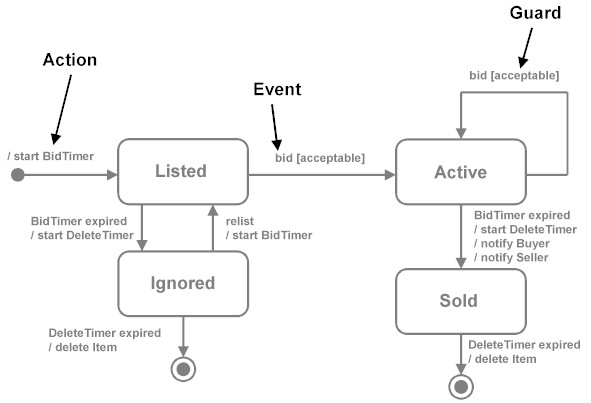
**Traceability Testing**   
Finally, after having our domain expert scour the class diagrams, we proceed to the third type of testing—traceability testing. We want to make certain that we can trace from the use cases and the sequence diagrams to the class diagrams and from the class diagrams back to the use cases and sequence diagrams. Again, we turn to one question: Is it consistent?

Consistent:

1. Is each object on the sequence diagram represented by a class on the class diagram?
2. Is every message on the sequence diagram reflected as a method in the appropriate class?

**State-Transition Diagrams**   
State-transition diagrams describe all of the states that an object can have, the events under which an object changes state (transitions), the conditions that must be fulfilled before the transition will occur (guards), and the activities undertaken during the life of an object (actions). State-transition diagrams are very useful for describing the behavior of individual objects over the full set of use cases that affect those objects. State-transition diagrams are not useful for describing the collaboration between objects that cause the transitions.

The UML notation for state-transition diagrams is shown below:



**Notation**   
For those not familiar with the notation used for state-transition diagrams, some explanation is in order.

***State***. A condition during the life of an object in which it satisfies some condition, performs some action, or waits for some event.

***Event***. An occurrence that may trigger a state transition. Event types include an explicit signal from outside the system, an invocation from inside the system, the passage of a designated period of time, or a designated condition becoming true.

***Guard***. A boolean expression which, if true, enables an event to cause a transition.

***Transition***. The change of state within an object.

***Action***. One or more actions taken by an object in response to a state change.

**Syntax Testing**   
Let’s begin with the simplest kind of testing—syntax testing. When performing syntax testing, we are verifying that the state-transition diagram contains correct and proper information. We ask three kinds of questions: Is it complete? Is it correct? Is it consistent?   
  
By now I’m sure you remember our secret from the previous articles. You do not need to know the answers to any of these questions before asking them. It is the process of asking and answering that is most important. Listen to the answers you are given. Are people confident about their answers? Can they explain them rationally? Or do they hem and haw and fidget in their chairs or look out the window or become defensive when you ask? Now for the questions:

Correct:

1. Does each state-transition diagram have one and only one initial state?
2. If the state-transition diagram is an open-loop, is there at least one terminal state?
3. If the state-transition diagram is a closed-loop, is it really? (Almost all are actually open-loop)
4. Does each state have at least one exit transition?
5. If multiple guards exist for a single event, are the guards mutually exclusive?
6. Does each state have exactly one transition for each possible event-guard combination?
7. Have all redundant or duplicate states or transitions been removed?
8. Are all states reachable?
9. Is every "real" state in the world represented by one and only one state on the diagram?
10. Is each state and transition clearly named?
11. Are all possible paths also valid paths?
12. Are all valid paths represented?

**Domain Expert Testing**   
After checking the syntax of the state-transition diagrams, we proceed to the second type of testing—domain expert testing. Again, we have two options: find a domain expert or attempt to become one. (The second approach is always more difficult than the first, and the first can be very hard.) Continuing, we ask three kinds of questions: Is it complete? Is it correct? Is it consistent?

Complete:

1. Are all of the required states, events, guards, transitions, and actions shown on the diagram?
2. Are all exceptional cases handled properly?

Correct:

1. Are we using state-transition diagrams only for classes that have complex, interesting behavior?
2. Does the diagram correctly represent the open-loop/closed-loop nature of the class?
3. Are all of the required states, events, guards, transitions, and actions properly defined?

Consistent:

1. Does a one-to-one correspondence exist between an object’s events and its methods?

**Traceability Testing**   
Finally, after having our domain expert scour the state-transition diagrams, we proceed to the third type of testing—traceability testing. We want to make certain that we can trace from the requirements to the state-transition diagrams and from the state-transition diagrams back to the requirements. Again, we turn to one question: Is it consistent?

Consistent:

1. Do all states, events, guards, transitions, and actions in the requirements appear in the state-transition diagram?

**Conclusion**   
This set of questions, based on syntax, domain expert, and traceability testing; and focused on completeness, correctness, and consistency; is designed to get you started testing in an area with which you may not be familiar.