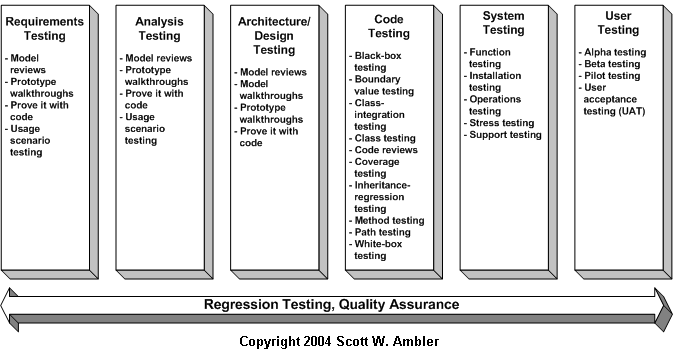
|  |  |
| --- | --- |
| The Full-Lifecycle Object-Oriented Testing (FLOOT) methodology is a collection of testing techniques to verify and validate object-oriented software. The FLOOT lifecycle is depicted in , indicating a wide variety of techniques (described in [Table 1](http://www.ambysoft.com/essays/floot.html#Table1)) are available to you throughout all aspects of software development.   The list of techniques is not meant to be complete – instead the goal is to make it explicit that you have a wide range of options available to you. It is important to understand that although the FLOOT method is presented as a collection of serial phases it does not need to be so: the techniques of FLOOT can be applied with evolutionary/agile processes as well.  The reason why I present the FLOOT in a “traditional” manner is to make it explicit that you can in fact test throughout all aspects of software development, not just during coding. |  |

**Figure 1. The FLOOT Lifecycle.**



**Table 1. Testing techniques.**

|  |  |
| --- | --- |
| **FLOOT Technique** | **Description** |
| Black-box testing | Testing that verifies the item being tested when given the appropriate input provides the expected results. |
| Boundary-value testing | Testing of unusual or extreme situations that an item should be able to handle. |
| Class testing | The act of ensuring that a class and its instances (objects) perform as defined. |
| Class-integration testing | The act of ensuring that the classes, and their instances, form some software perform as defined. |
| Code review | A form of technical review in which the deliverable being reviewed is source code. |
| Component testing | The act of validating that a component works as defined. |
| Coverage testing | The act of ensuring that every line of code is exercised at least once. |
| Design review | A technical review in which a design model is inspected. |
| Inheritance-regression testing | The act of running the test cases of the super classes, both direct and indirect, on a given subclass. |
| Integration testing | Testing to verify several portions of software work together. |
| Method testing | Testing to verify a method (member function) performs as defined. |
| Model review | An inspection, ranging anywhere from a formal technical review to an informal walkthrough, by others who were not directly involved with the development of the model. |
| Path testing | The act of ensuring that all logic paths within your code are exercised at least once. |
| Prototype review | A process by which your users work through a collection of use cases, using a prototype as if it was the real system. The main goal is to test whether the design of the prototype meets their needs. |
| Prove it with code | The best way to determine if a model actually reflects what is needed, or what should be built, is to actually build software based on that model that show that the model works. |
| Regression testing | The acts of ensuring that previously tested behaviors still work as expected after changes have been made to an application. |
| Stress testing | The act of ensuring that the system performs as expected under high volumes of transactions, users, load, and so on. |
| Technical review | A quality assurance technique in which the design of your application is examined critically by a group of your peers. A review typically focuses on accuracy, quality, usability, and completeness. This process is often referred to as a walkthrough, an inspection, or a peer review. |
| Usage scenario testing | A testing technique in which one or more person(s) validate a model by acting through the logic of usage scenarios. |
| User interface testing | The testing of the user interface (UI) to ensure that it follows accepted UI standards and meets the requirements defined for it. Often referred to as graphical user interface (GUI) testing. |
| White-box testing | Testing to verify that specific lines of code work as defined. Also referred to as clear-box testing. |

I’d like to share a few of my personal philosophies with regards to testing:

1. **The goal is to find defects**.  The primary purpose of testing is to validate the correctness of whatever it is that you’re testing.  In other words, successful tests find bugs.
2. **You can validate all artifacts**.  As you will see in this chapter, you can test all your artifacts, not just your source code. At a minimum you can review models and documents and therefore find and fix defects long before they get into your code.
3. **Test often and early**.  The potential for the [cost of change](http://www.agilemodeling.com/essays/costOfChange.htm) to rise exponentially motivates you to test as early as possible.  [Figure 2](http://www.ambysoft.com/essays/floot.html#Figure2) compares the average cost to address defects found by various agile and traditional approaches to testing.  The goal is to reduce the feedback cycle as much as possible.
4. **Testing builds confidence**.  In [Extreme Programming Explained](http://www.amazon.com/exec/obidos/ASIN/0201616416/ambysoftinc) Kent Beck makes an interesting observation that when you have a full test suite, a test suite is a collection of tests, and if you run it as often as possible, that it gives you the courage to move forward.  Many people fear making a change to their code because they’re afraid that they’ll break it, but with a full test suite in place if you do break something you know you’ll detect it and then fix it.
5. **Test to the risk of the artifact**.  The riskier something is, the more it needs to be reviewed and tested. In other words you should invest significant effort testing in an air traffic control system but nowhere near as much effort testing a “Hello World” application.
6. **One test is worth a thousand opinions**.  You can tell me that your application works, but until you show me the test results, I will not believe you.

**Figure 2. Comparing the average cost of change of various defect finding techniques.**