The approach for the application service tools and objects were designed around specifically testing for the parameters that the different variables needed to meet. The moment the object was created for Contact, Appointment, or Task, every single variable was checked to make sure that they fit the criteria necessary. That was that they weren’t too long, nor were they null. In the case of the Contact object it even had an extra criteria of the phone number needing to be exactly 10 characters long. As such it was tested for a slightly different criteria than everything else was tested for.

For example First name was tested as such:  
 if (firstName == null || firstName.length() > 10) {

throw new IllegalArgumentException("Invalid first name");

}

And Number was tested in almost the same manner except for the fact that it’s checking if number is NOT 10:

if (number == null || number.length() != 10) {

throw new IllegalArgumentException("Invalid number");

}

In either case, if the criteria is not met, then the program will throw an error exception and cancel the creation of the object at all, thus not wasting memory on an object that is invalid. The JUnit tests that I performed also were particularly effective in testing the validity of the code in that they tested against these exact vulnerabilities. They tested against whether the object’s variables were correctly formatted. Then they were tested to see if the wrong inputs were inputted in both cases, whether too long, too short, or null, and tested to see if there would be an exception thrown. Some examples of this testing can be seen here, where specifically the first name of the contact is tested to see if an exception is thrown when the value is too long.

void testFirstNameTooLong() {

Assertions.assertThrows(IllegalArgumentException.class, ()->{

new Contact("Thisnametoolong", "Last Name", "5556667777", "101 test st", "1");

});

}

Then testing if first name is null:  
void testFirstNameIsNull() {

Assertions.assertThrows(IllegalArgumentException.class, ()->{

new Contact(null, "Last Name", "5556667777", "101 test st", "1");

});

}

The Test coverage of the classes further supported this fact by showing that the logic and branching of the logic in the code was tested at a percentage higher than 85%, in most cases it was 100% of the code. This details that the majority of the cases of the function of the code was technically sound and correct. Nothing is ever a guarantee but by testing through these various branches of potential logic, we can ensure that it is majoritively correct. Efficiency was also obtained by using the shortest amount of code possible for many of the cases. This can be pointed back to the testing if the number variable met criteria. Instead of testing if it was larger than 10 AND shorter than 10, we avoid the unnecessary extra logic steps and simply check to see if the number is NOT 10 digits long. Which is exactly the criteria in the first place.

The testing technique that was used was unit testing. Unit testing is where you’re testing the specific parts of the program. This is the kind of testing that is used to make sure that a ‘unit’ of the program is functioning as intended. By testing the various variables for correct and incorrect input we can verify that the different situations that need to be avoided and/or allowed to happen are functioning properly as intended. As well as testing the section of the program to work as according to the requirements of the program.

There are various other types of testing techniques as well in the form of White box and Black box testing. Black box testing being where you are testing it as a user, you have no idea how the program is functioning internally and therefore you’re testing it from the experience of using the program first hand. You’re just inputting things and expecting the program to either accept them or deny them and testing how and why it does what it does based on your uninformed input. Whereas white box testing is exactly the opposite.

In white box testing, you know the internal structure of the program and know how it is intended to work. As such you can test and push at the boundaries of the program and test that it is functioning in the way that it is intended and needed to function. They generally will observe and monitor the interactions of the different units of a program together and see how they interact and why they respond in the ways they do. This is done to make sure that each unit of the program is working together in tandem in a correct nad functioning manner as per their requirements of the program and with minimal bugs and errors while doing so.

It is important to understand these tests and furthermore understand that your bias can play a heavy hand in creating “blind spots” when it comes to testing your code. Think of times when you’re doing a problem, and you do something wrong, some small detail is missed, and no matter how often you go over the problem, you can’t figure out what’s wrong. Sometimes someone simply points out a small thing and you realize your folly. Sometimes you can come back to the problem hours or even days later and then recognize what you did wrong when you get a fresh perspective and mind set on the subject. Sometimes we get so focused on believing that we’re right, that we miss what we did wrong in a situation and often times it can be useful to have another person look over what we’ve done because a lot of times the mistake can be fantastically small. On that same note it’s always important to approach tests with the mindset that you can and probably will definitely be wrong. By having this mindset, it can open you up to better seeing the small mistakes that can lead to bigger errors and/or to fix them in the early stages before they become bigger mistakes. It’s never a good thing to just assume your code is correct, rather it’s better to assume your code is incorrect so that you can more thoroughly and meticulously look through it to find your potential errors.