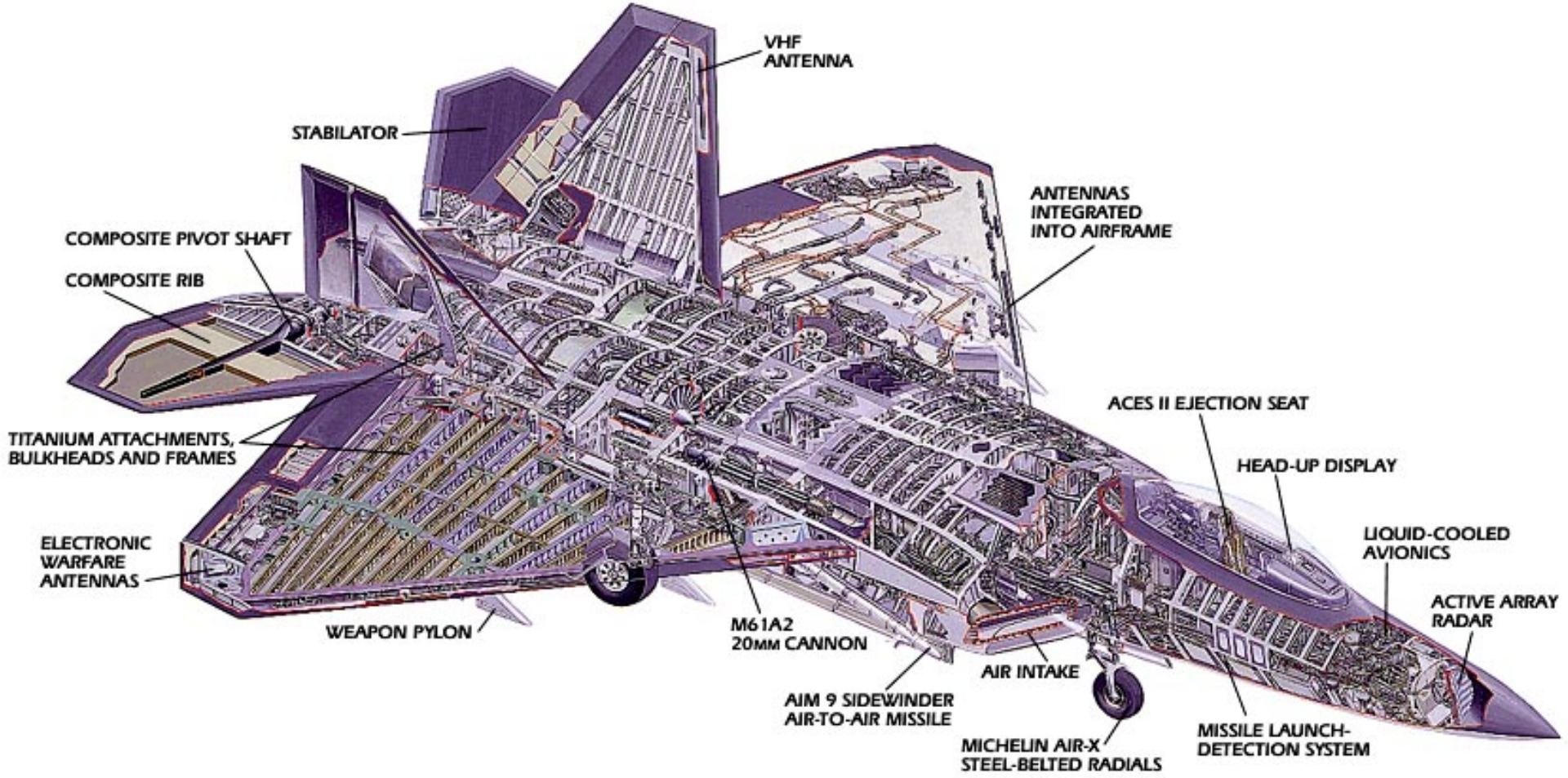
Unit Testing &
Defensive Programming

F-22 Raptor Fighter

F-22 Raptor Fighter

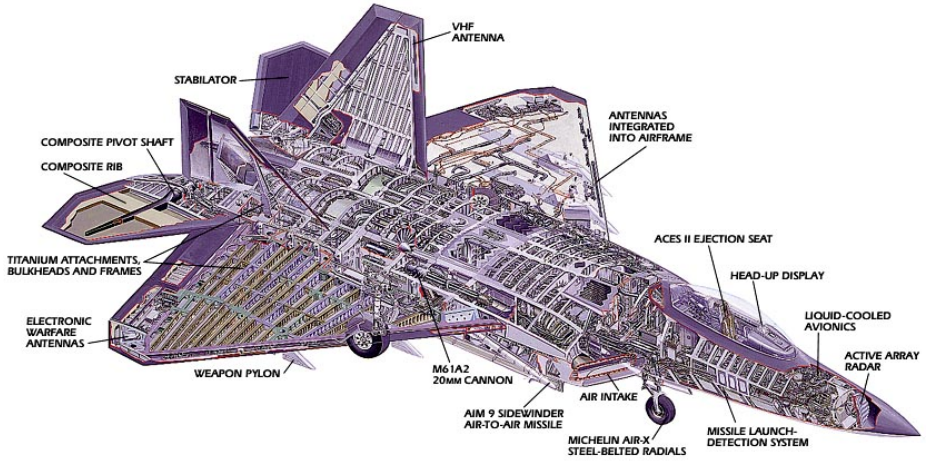
- Manufactured by Lockheed Martin & Boeing
- How many parts does the F-22 have?



F-22 Raptor Fighter

- What would happen if Lockheed assembled an F-22 with "untested" parts (i.e., parts that were built but never verified)?

- It wouldn't work, and in all likelihood you would never be able to make it work
  - Cheaper and easier to just start over



Managing implementation complexity

- Individual parts should be verified before being integrated with other parts
- Integrated subsystems should also be verified
- If adding a new part breaks the system, the problem must be related to the recently added part
- Track down the problem and fix it
- This ultimately leads to a complete system that works

2 approaches to programming

- Approach #1
  - "I wrote ALL of the code, but when I tried to compile and run it, nothing seemed to work!“
- Approach #2
  - Write a little code (e.g., a method or small class)
  - Test it
  - Write a little more code
  - Test it
  - Integrate the two verified pieces of code
  - Test it
  - …

Unit testing

- Large programs consist of many smaller pieces
  - Classes, methods, packages, etc.
- "Unit" is a generic term for these smaller pieces
- Three important types of software testing are:
  - Unit Testing (test units in isolation)
  - Integration Testing (test integrated subsystems)
  - System Testing (test entire system that is fully integrated)
- Unit Testing is done to test the smaller pieces in isolation before they are combined with other pieces
  - Usually done by the developers who write the code

What unit tests do

- Unit tests create objects, call methods, and verify that the returned results are correct
- Actual results vs. Expected results
- Unit tests should be automated so that they can be run frequently (many times a day) to ensure that changes, additions, bug fixes, etc. have not broken the code
  - Regression testing
- Notifies you when changes have introduced bugs, and helps to avoid destabilizing the system

Test driver program

- The tests are run by a "test driver", which is a program that just runs all of the unit test cases
- It must be easy to add new tests to the test driver
- After running the test cases, the test driver either tells you that everything worked, or gives you a list of tests that failed
- Little or no manual labor required to run tests and check the results
- Tools like Ant or Make are often used to automate the building and running of the test driver (e.g., `$ ant test`)

JUnit testing framework

- JUnit is a “framework” for implementing automated unit tests
- JUnit Documentation
- Example: Contact Manager unit tests

JUnit testing framework

- For every class in the “src” directory, create a corresponding class in the “test” directory
  - src/shared/model/Contact.java =>
    test/shared/model/ContactTest.java
  - src/server/database/Database.java =>
    test/server/database/DatabaseTest.java
  - src/server/database/Contacts.java =>
    test/server/database/ContactsTest.java
  - src/client/communication/ServerCommunicator.java =>
    test/client/communication/ServerCommunicatorTest.java

- Import JUnit classes
  - import org.junit.*;
  - import static org.junit.Assert.*;

JUnit testing framework

- Use JUnit annotations to mark test methods

Annotation	Description
@Test public void method()	The annotation @Test identifies that a method is a test method.
@Before public void method()	Will execute the method before each test. This method can prepare the test environment (e.g. read input data, initialize the class).
@After public void method()	Will execute the method after each test. This method can cleanup the test environment (e.g. delete temporary data, restore defaults).

JUnit testing framework

- Use JUnit annotations to mark test methods

Annotation	Description
@BeforeClass public void method()	Will execute the method once, before the start of all tests. This can be used to perform time intensive activities, for example to connect to a database.
@AfterClass public void method()	Will execute the method once, after all tests have finished. This can be used to perform clean-up activities, for example to disconnect from a database.
@Test (expected = Exception.class)	Fails, if the method does not throw the named exception.
@Test(timeout=100)	Fails, if the method takes longer than 100 milliseconds.

JUnit testing framework

- Use JUnit `assert*` methods to implement test cases

- JUnit Assert Method Documentation

Running JUnit tests

- Run them in Eclipse
  - Right-Click on any test class, package, folder, or project in the Package Explorer
  - Select Run As => JUnit Test from the menu
  - This will run all test methods in the selected class, package, folder, or project

Running JUnit tests

- Create a test driver program to run the tests
  - test/main/UnitTestDriver.java in Contact Manager

```
package main;

public class UnitTestDriver {

    public static void main(String[] args) {

        String[] testClasses = new String[] {
            "shared.model.ContactTest",
            "server.database.DatabaseTest",
            "server.database.ContactsTest",
            "client.communication.ServerCommunicatorTest"
        };

        org.junit.runner.JUnitCore.main(testClasses);
    }

}
```

Defensive Programming

- Good programming practices that protect you from your own programming mistakes, as well as those of others
  - Assertions
  - Parameter Checking

Assertions

- As we program, we make many assumptions about the state of the program at each point in the code
  - A variable's value is in a particular range
  - A file exists, is writable, is open, etc.
  - Some data is sorted
  - A network connection to another machine was successfully opened
  - …
- The correctness of our program depends on the validity of our assumptions
- Faulty assumptions result in buggy, unreliable code

Assertions

```
int binarySearch(int[] data, int searchValue) {

   // What assumptions are we making about the parameter values?

   …
}
```

- data != null
- data is sorted

- What happens if these assumptions are wrong?

Assertions

- Assertions give us a way to make our assumptions explicit in the code
- `assert temperature > 32 && temperature < 212;`
- The parameter to assert is a boolean condition that should be true
- `assert condition;`
- If the condition is false, Java throws an AssertionError, which crashes the program
- Stack trace tells you where the failed assertion is in the code

Assertions

```
int binarySearch(int[] data, int searchValue) {

   assert data != null;
   assert isSorted(data);

   …
}
String[] someMethod(int y, int z) {

   assert z != 0;
   int x = y / z;

   assert x > 0 && x < 1024;
   return new String[x];
}
```

Assertions

- Assertions are little test cases sprinkled throughout your code that alert you when one of your assumptions is wrong
- This is a powerful tool for avoiding and finding bugs

- Assertions are usually disabled in released software

- In Java, assertions are DISABLED by default
- To turn enable them, run the program with the `–enableassertions` (or `-ea`) option

- java –enableassertions MyApp
- java –ea MyApp

- In Eclipse, the `–enableassertions` option can be specified in the **VM arguments** section of the **Run Configuration** dialog

Assertions

- Alternate form of assert
- `assert condition : expression;`
- If condition is false, expression is passed to the constructor of the thrown AssertionError

```
int binarySearch(int[] data, int searchValue) {

   assert data != null : ”binary search data is null”;
   assert isSorted(data) : ”binary search data is not sorted”;
   …
}
String[] someMethod(int y, int z) {

   assert z != 0 : ”invalid z value”;
   int x = y / z;

   assert x > 0 && x < 1024 : x;
   return new String[x];
}
```

Assertions

- If one of my assumptions is wrong, shouldn't I throw an exception?

- No. You should fix the bug, not throw an exception.

Parameter Checking

- Another important defensive programming technique is "parameter checking"
- A method or function should always check its input parameters to ensure that they are valid
- If they are invalid, it should indicate that an error has occurred rather than proceeding
- This prevents errors from propagating through the code before they are detected
- By detecting the error close to the place in the code where it originally occurred, debugging is greatly simplified

Parameter Checking

- Two ways to check parameter values
  - assertions
  - if statement that throws exception if parameter is invalid

```
int binarySearch(int[] data, int searchValue) {
   assert data != null;
   assert isSorted(data);
   …
}

int binarySearch(int[] data, int searchValue) {
   if (data == null || !isSorted(data)) {
      throw new InvalidArgumentException();
   }
   …
}
```

Parameter Checking

- Should I use assertions or if/throw to check parameters?

- If you have control over the calling code, use assertions
  - If parameter is invalid, you can fix the calling code

- If you don't have control over the calling code, throw exceptions
  - e.g., your product might be a class library that is called by code you don’t control