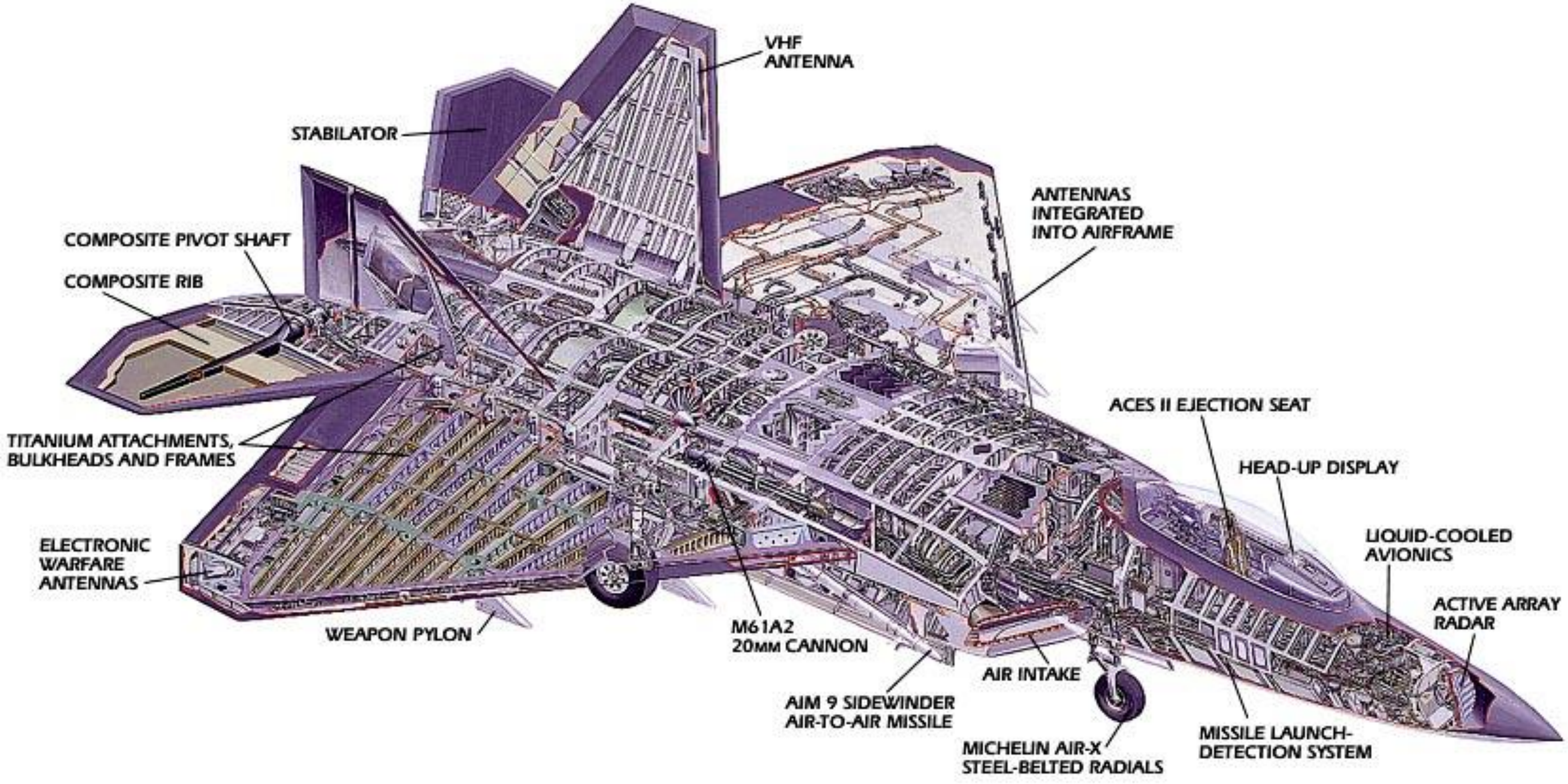
Unit Testing

F-22 Raptor Fighter

F-22 Raptor Fighter

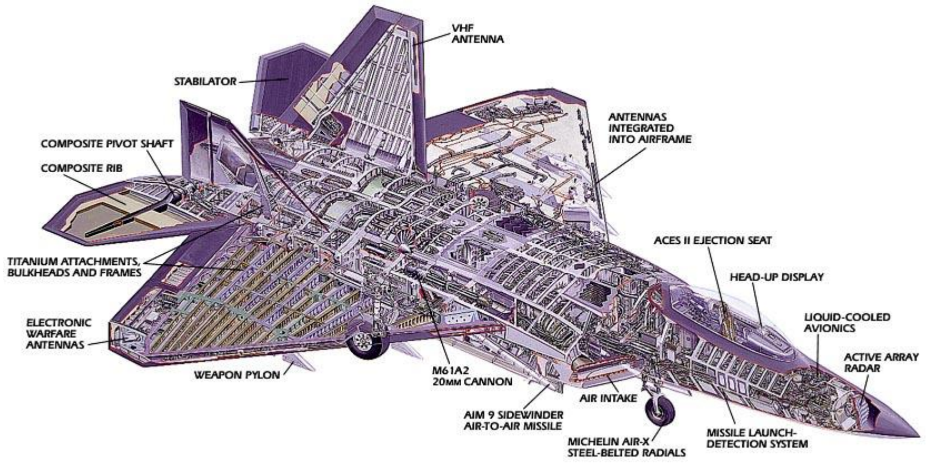
- Manufactured by Lockheed Martin & Boeing
- How many parts does the F-22 have?


VHF ANTENNA
STABILATOR
ANTENNAS INTEGRATED INTO AIRFRAME
COMPOSITE PIVOT SHAFT
COMPOSITE RIB
ACES II EJECTION SEAT
TITANIUM ATTACHMENTS, BULKHEADS AND FRAMES
HEAD-UP DISPLAY
LIQUID-COOLED AVIONICS
ELECTRONIC WARFARE ANTENNAS
ACTIVE ARRAY RADAR
WEAPON PYLON
M61A2 20MM CANNON
AIR INTAKE
AIM 9 SIDEWINDER AIR-TO-AIR MISSILE
MICHELIN AIR-X STEEL-BELTED RADIALS
MISSILE LAUNCH-DETECTION SYSTEM

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


VHF ANTENNA
STABILATOR
ANTENNAS INTEGRATED INTO AIRFRAME
COMPOSITE PIVOT SHAFT
COMPOSITE RIB
ACES II EJECTION SEAT
TITANIUM ATTACHMENTS, BULKHEADS AND FRAMES
HEAD-UP DISPLAY
LIQUID-COOLED AVIONICS
ELECTRONIC WARFARE ANTENNAS
ACTIVE ARRAY RADAR
WEAPON PYLON
M61A2 20MM CANNON
AIR INTAKE
AIM 9 SIDEWINDER AIR-TO-AIR MISSILE
MICHELIN AIR-X STEEL-BELTED RADIALS
MISSILE LAUNCH-DETECTION SYSTEM

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 units)
  - 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

Android testing framework

- Android provides a framework for writing automated unit tests
  - Based on the popular JUnit unit testing framework

- There are two types of Android unit tests
  - Local Unit Tests
    - These tests depend only on standard Java classes, and so can be run on the development computer instead of on an Android device
    - You will create local unit tests for the Family Map Server project
  - Instrumented Unit Tests
    - These tests depend on Android-specific classes, and so must be run on an Android device
    - You will create instrumented unit tests for the Family Map Client project

Android local unit tests

- Official Documentation
- Can run on the development computer without a device or emulator
- Module's primary source code is located in the folder
  - `<module>/src/`**`main`**`/java/<package>`
- Local unit test code is located in the folder
  - `<module>/src/`**`test`**`/java/<package>`

Android local unit tests

- Example: junit-example (on web site)
- “spellcheck” module contains code for web-based spelling checker
- “Real” classes are in:
  - src/**main**/java/spellcheck/*.java
  - src/**main**/java/dataaccess/*.java
- “Test” classes are in:
  - src/**test**/java/spellcheck/*.java
  - src/**test**/java/dataaccess/*.java

Android local unit tests

- Local test classes are written using the JUnit 4 unit test framework
- Include the following in app/build.gradle

```
dependencies {
   …
   testCompile 'junit:junit:4.12'
}
```

- Import JUnit 4 classes

```
import org.junit.*;
import static org.junit.Assert.*;
```

Android local unit tests

- Test classes are just regular classes (no special superclass)
- Test methods may have any name (need not be test*), but must have the @Test annotation on them
- Common initialization code can be placed in a method (any name) with the @Before annotation
- Common cleanup code can be placed in a method (any name) with the @After annotation
- Use JUnit `assert*` methods to implement test cases
- JUnit 4 Assert Method Documentation

Running local unit tests (from Android Studio)

- No device or emulator is needed
- To run a single test class, in the “Project” tool window right-click on a test class name, and select “Run Tests” or “Debug Tests”
- To run all of your local unit tests, right-click on the “test/java” folder, and select “Run All Tests” or “Debug All Tests”

Running local unit tests (from command-line)

- Write a test driver class whose “main” method invokes the org.junit.runner.JUnitCore class to run your unit tests

- Run your test driver program from the command-line:

```
java –cp build\classes\main;build\classes\test;libs\junit-4.12.jar;libs\hamcrest-all-1.3.jar;libs\sqlite-jdbc-3.16.1.jar TestDriver
```

- For the Family Map Server project, you will create a bash shell script that will compile and run your unit tests from the command-line

JUnit 4 unit testing framework

- JUnit 4 Documentation
- Use JUnit 4 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 4 unit testing framework

- Use JUnit 4 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.

Database Unit Tests

- When writing unit tests for your database code, there are additional things to think about
- Put database driver JAR file on the class path
- Each unit test should start with a pristine database so prior tests have no effect
  - Can re-create tables before each test
  - Or, you can “rollback” the effects of each test so they are undone and don’t affect later tests