Introduction

CIS-2730, Software Engineering Projects
Vermont Technical College
Peter C. Chapin
(Last Revised: January 2022)

What is Ada?

- Brief history…
  - Developed in the early 1980s by the DoD in response to problems with their software development process
    - Software was late, too expensive, not reusable, didn’t work
    - Developed requirements for a better programming language
    - Decided no existing language satisfied those requirements: created Ada
  - Ada mandated by the DoD. This turned out to be a bad idea
    - Early Ada compilers were expensive and of poor quality
    - DoD dropped the Ada mandate in the 1990s. Left a bad taste for Ada

Ada Today

- Fast forward to 2022…
  - – Ada 1984
    - Original version. Far ahead of its time.
  - – Ada 1995
    - Added OOP. Added “protected objects” (related to concurrency).
  - – Ada 2005
    - Added Java-like Interfaces. Merged OOP and concurrent programming.
  - – Ada 2012
    - Added contracts.

Ada 2022

- New standard about to be released…
  - Adds support for parallel constructs
  - Numerous other fixes/enhancements
- New standard released every 5-10 years.
  - Next version circa 2030?
- Modern Ada compilers are good
  - Fast, efficient, good error messages
  - No different than compilers in other languages
  - Yet many people still think of Ada as “old” or “a mistake.”

Ada is Conservative

- The standard evolves slowly
  - Many proposals for new features are rejected
  - Only features that address Ada’s design goals (next slides) and that are being asked for by the user community are seriously considered.
- “There are many fine programming language features, but that doesn’t make them right for Ada.”
- Backward compatibility is a major goal
  - Can still (usually) compile Ada84 programs with modern compilers.

What is Ada Good For? (Part 1)

- Programming in the large
  - Multi-million line programs written by multiple, loosely associated teams. Ada has features to ensure consistency of separately developed components.
- Reliable programming
  - For safety-critical and mission-critical applications where software failure means loss of life, money, and reputation. Ada has features to catch certain errors early in the development lifecycle.

What is Ada Good For? (Part 2)

- Programming for the long term
  - Programs that will be used for decades. Ada has features to enhance the readability and understandability of programs.
- Embedded systems programming
  - Programs that control machines, often with real-time requirements. Ada has features for low level control, concurrency, and real-time constructs.

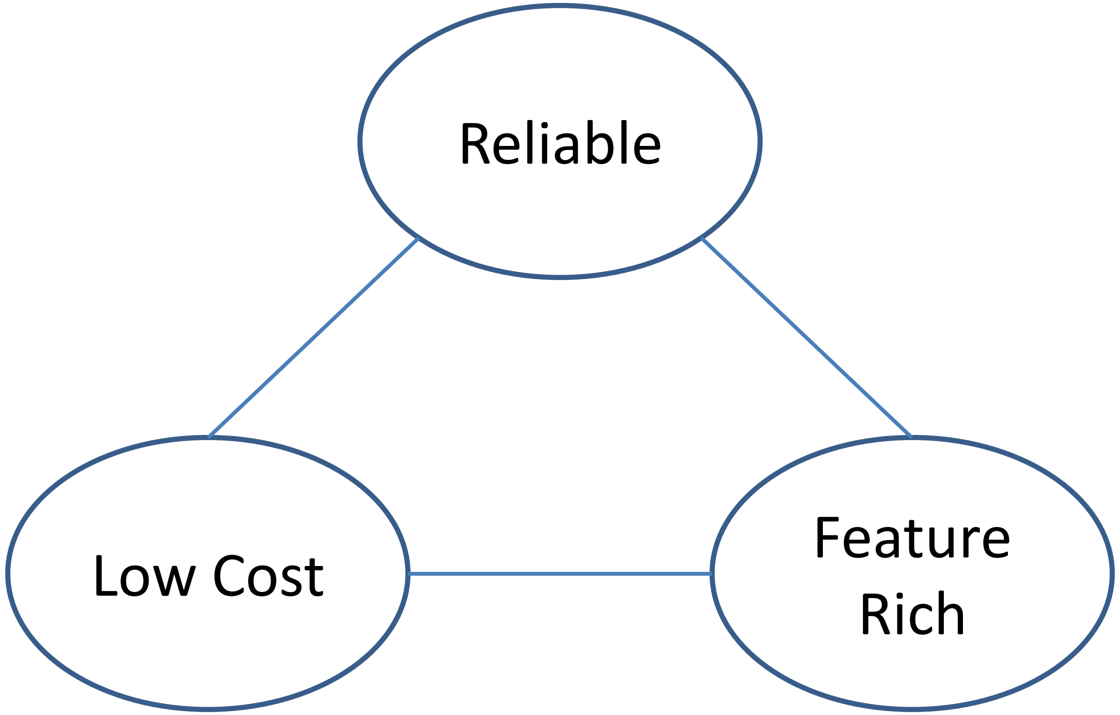
Ada as a Software Engineer’s Language

- Ada’s design encourages well-constructed programs
  - Some (bad) program designs just won’t compile!
  - The most natural way to use the language tends to produce well designed programs.
- Ada will change the way you think about programming
  - … *and make you a better programmer in any language*!

What about SPARK?

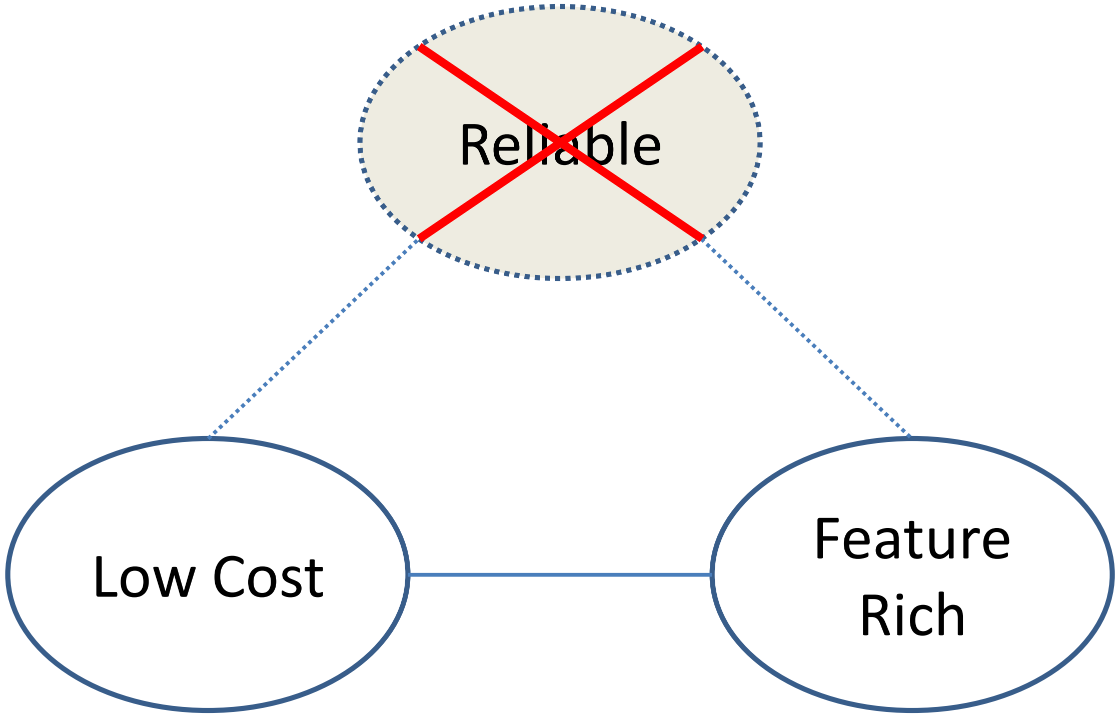
- Ada was designed for safety-critical and mission-critical apps
- SPARK further increases the reliability of Ada programs
  - Studies show that SPARK is about 10x more reliable than Ada
  - … and that Ada is about 10x more reliable than C.
- How?
  - By introducing the possibility of proof

Three Attributes



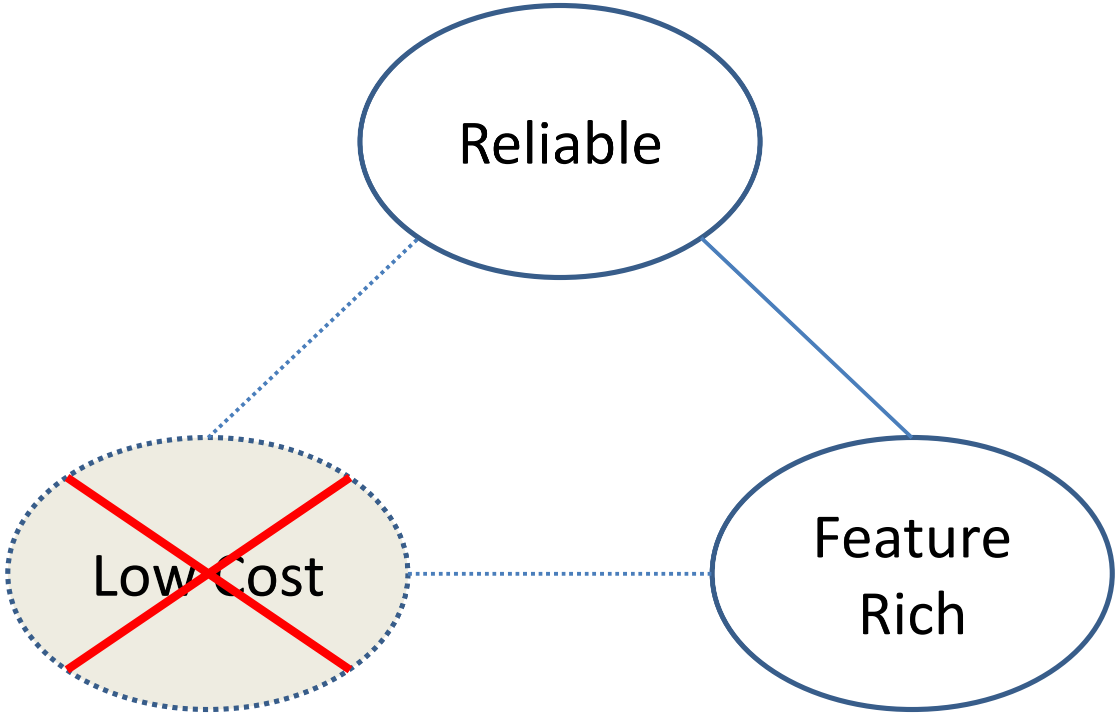
Can’t have all three

Sacrifice Reliability



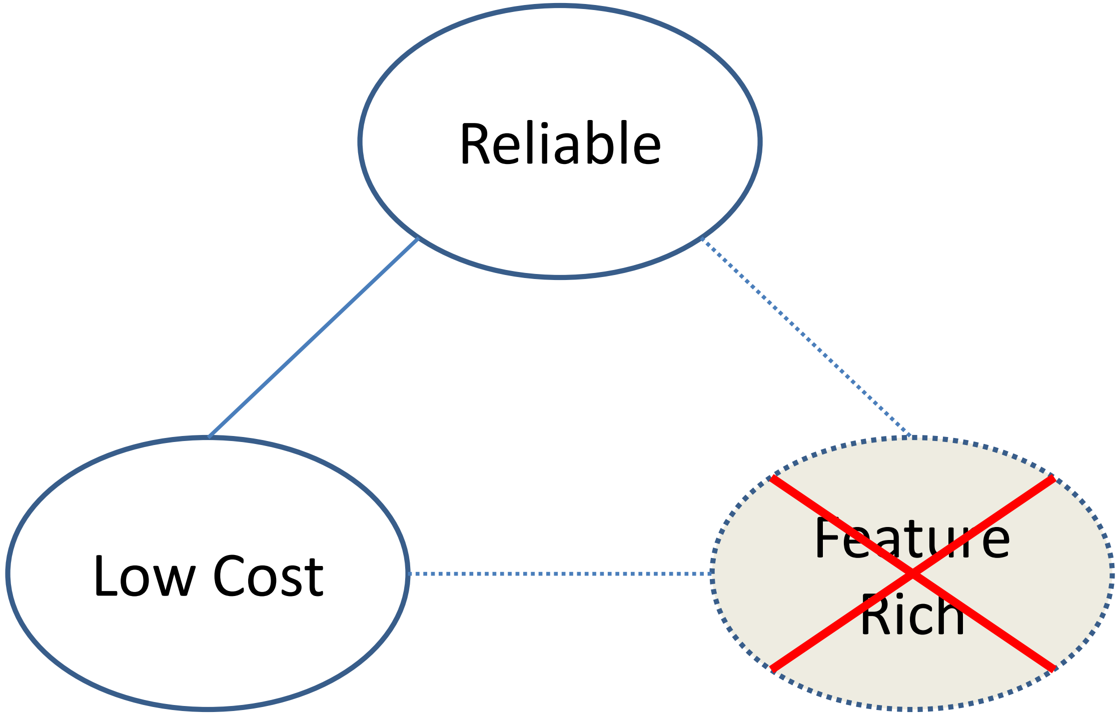
Cheap, feature rich, doesn’t work

Sacrifice Low Cost



Lots of features that work, expensive

Sacrifice Features



Works well and is cheap, but doesn’t do much

Broken Software

- Most software has faults (bugs)
  - As long as it works “well enough” we are happy
  - Cost of removing every last fault can be extreme
    - In general it is impossible anyway
- Some software requires very high reliability
  - … *transcending all other considerations*

Safety Critical Software

- If the software fails, people die
  - Medical
  - Avionics
  - Automotive
  - Military
  - Industrial
  - Nuclear

Mission Critical Software

- If the software fails, Bad Things happen
  - Financial Transactions
  - Backbone network switches
  - Robotic spacecraft

Failure causes massive lose of time, money, or reputation

Security Sensitive Software

- If the software fails, security is compromised
  - Unauthorized access
  - Unauthorized modification
  - Unauthorized control
  - Exposure of private information
  - Cyber war

Failure is exploited by an attacker to break security contracts

High Integrity Software

- Catch-all Term…
  - *Safety Critical*
  - *Mission Critical*
  - *Security Sensitive*

High Integrity Programming is about writing such software

Two Approaches

- Testing
  - – Exercise the software to verify correct operation
    - Industry standard approach
    - VTC has Q & A course that covers this
- Formal Methods
  - – Use mathematics to prove the software correct
    - More difficult
    - Used by the high integrity developers
    - Less perfected… still being actively researched

Testing Not Exhaustive

```
function Sum(X : Integer, Y : Integer) return Integer is
begin
  return X + Y;
end Sum;
```

- With 32-bit integers there are $2^{32} * 2^{32} = 2^{64}$ test cases
- Who checks them all?
- In fact, many of them don’t “work” (overflow)
- Many subprograms have infinitely many test cases

Testing

- Testing is all about finding good test cases
  - Using only 0.000001% of possible test cases…
    - Find 99.99% of all faults!
  - Surprisingly this can work
    - You must choose test cases with care
- What about the last 0.01% faults?
  - Good enough?
  - *Not if your life depends on it*

Zune 30 Phone Failure

```
year = ORIGINYEAR; /* = 1980 */

while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}
```

Someone forgot to test what this does on the last day of a leap year.

… as a result the Zune phone failed to boot on December 31, 2008.

Fortunately nobody died because their phone didn’t boot.

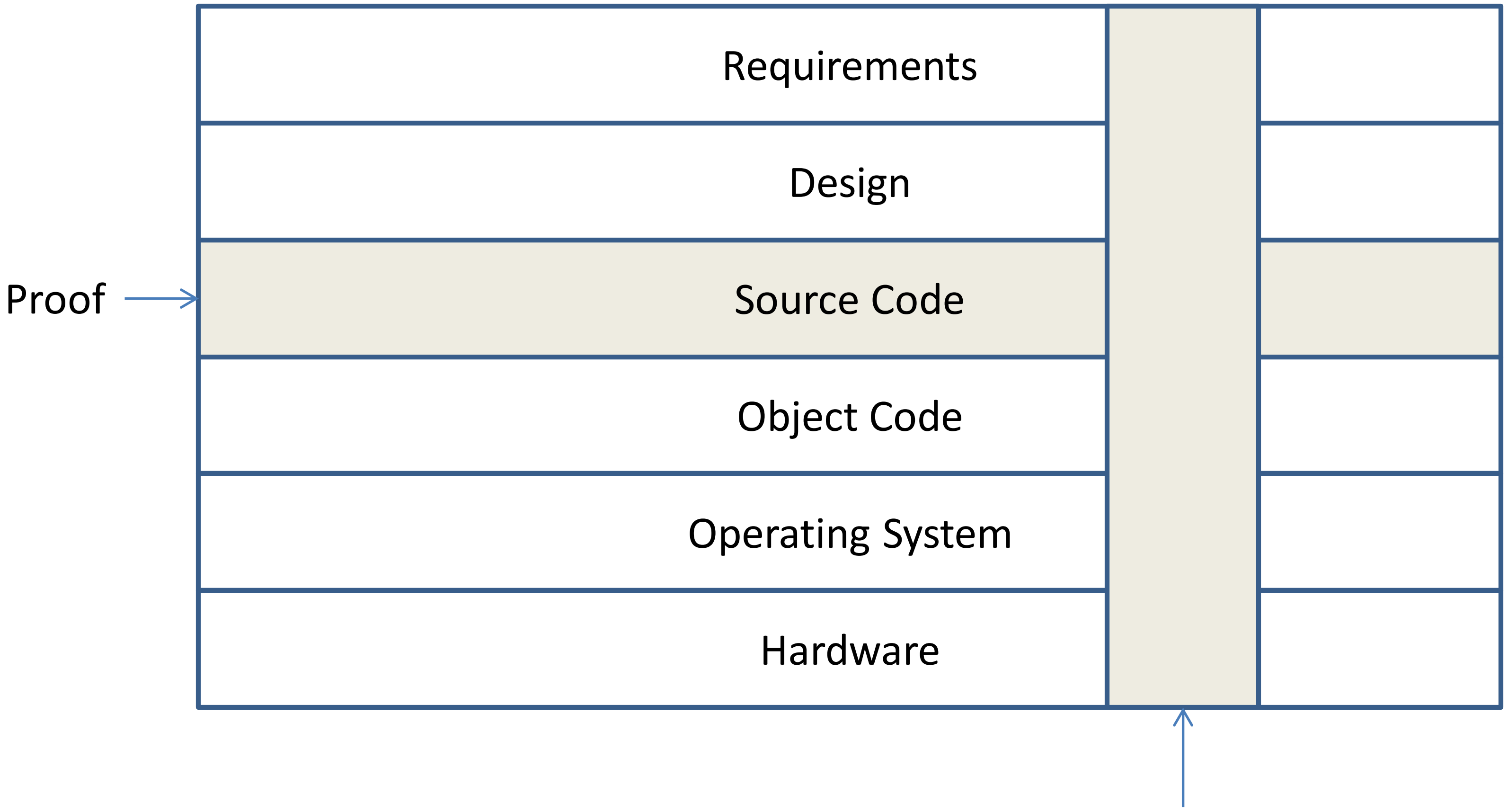
http://www.zuneboards.com/forums/showthread.php?t=38143

Proof *is* Exhaustive

- You can prove things about infinite objects
  - “There are infinitely many prime numbers.”
    - You don’t need to test them all.
- **Formal Methods**…
  - … is about applying techniques of mathematical proof to programming.
  - “This function computes the correct result in *all* cases.”
    - You don’t need to test them all.

So if I prove my program correct, I don’t have to test it.

Wrong!



Requirements

Design

Proof

Source Code

Object Code

Operating System

Hardware

Notice all the white space!

Testing

Testing + Proof

- Testing and proof complement each other
  - *Work together to cover all situations*
- Some things very hard to prove
  - *Let those gaps guide your testing*
- Some things very hard to test
  - *Explore those areas with proof*

Alaskan Ice Buoy

```
if Latitude_Degrees <= 90 and
   Latitude_Minutes <  60 then

   Total_Latitude_Minutes :=
      60 * Latitude_Degrees + Latitude_Minutes;
end if;
```

- `Latitude_Degrees` and `Latitude_Minutes` come from GPS device
- SPARK tools unable to prove that `Total_Latitude_Minutes` in range (0 .. 5400)

Alaskan Ice Buoy

Input validation wrong!
Allows 90^o 59′

```
if Latitude_Degrees <= 90 and
   Latitude_Minutes <  60 then

   Total_Latitude_Minutes :=
      60 * Latitude_Degrees + Latitude_Minutes;
end if;
```

- Problem only arises if GPS defective *and* produces a latitude between 90^o and 91^o.

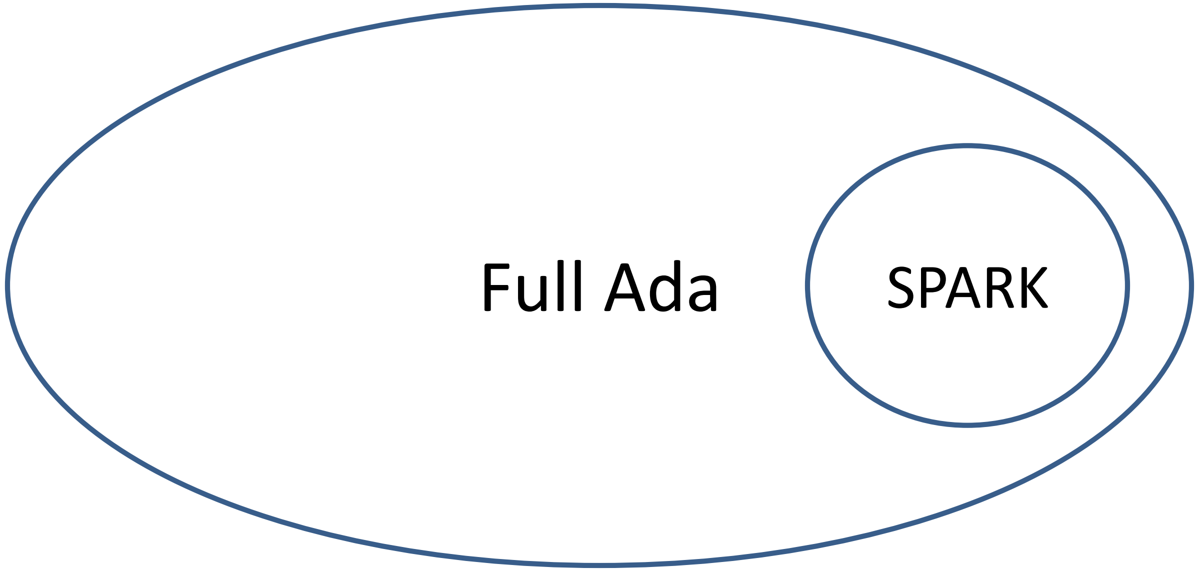
It could happen… but testing probably won’t find it

SPARK

- SPARK Language
  - Subset of Ada for high integrity programming
- SPARK Tools
  - “Examine”
    - Verifies you are using the SPARK language
    - Analyzes information flow (no use of uninitialized values, all results are used)
  - “Prove”
    - Shows that no runtime error will ever occur and that all contracts will always be obeyed.

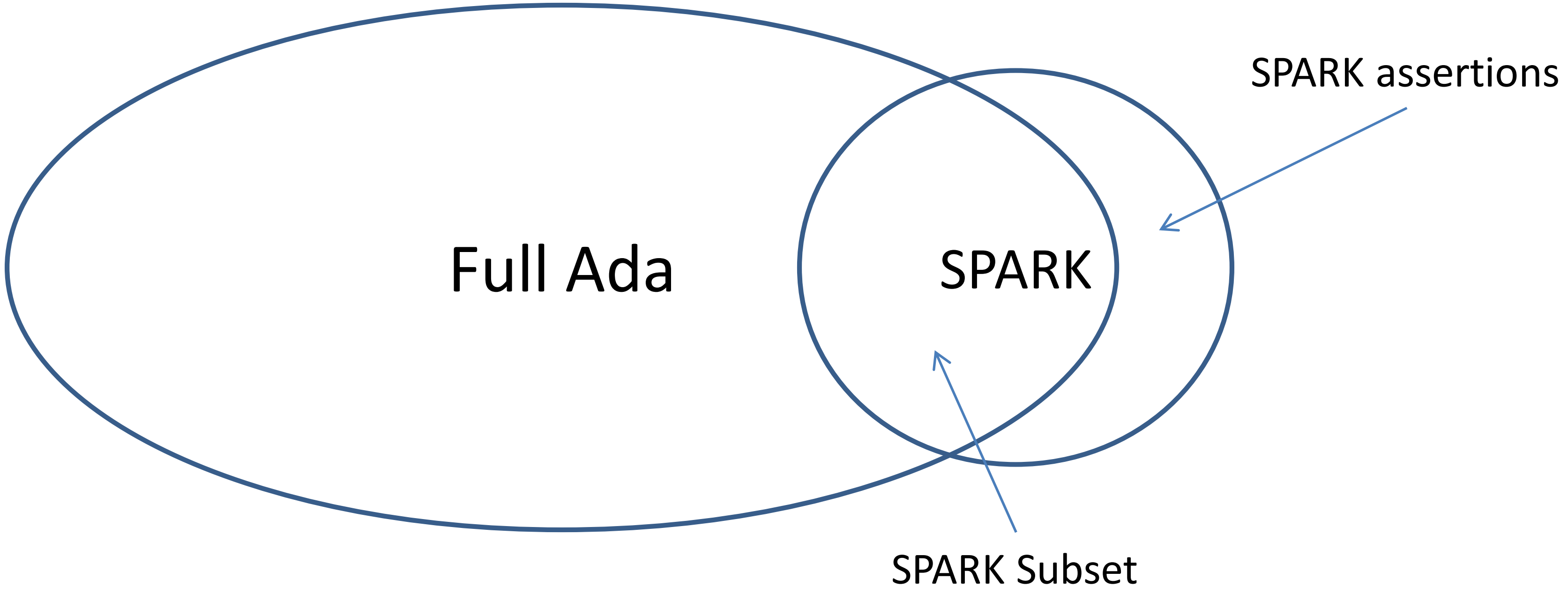
SPARK

- SPARK language is a subset of Ada
  - Not a compiler… relies on an Ada compiler



Assertions

- SPARK adds “assertions” to Ada
  - Ada has some assertions already
  - SPARK augments them with some additional ones
  - Proves that no assertion will ever fail



Separate Analysis

- SPARK program can be separate analyzed
  - Procedure bodies need not exist when code that uses them is analyzed
    - Annotations used when analyzing client code
    - Annotations checked when analyzing implementation
  - Parts of the program can be in full Ada
    - Use SPARK where needed
    - Use full Ada otherwise

Plan

- Rough outline…
  - Start with basic Ada
    - It’s useful to compile your code first as Ada
    - Less restrictive, better error messages
  - Update to SPARK + flow aspects
    - This will find some problems
  - Proofs of “freedom from run time error”
    - Prove the code exception free
  - Proofs of other correctness properties (contracts)

Sample Application

- Thumper
  - A secure time stamping service
  - Proof of freedom from exceptions means:
    - No “buffer overflows” (or equivalent)
    - Server can’t be brought down by misbehaving clients or internal errors
  - Will use crypto and some network communication

Have Fun!