Lecture 2 — Software Correctness

David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Software Correctness

RECOMMENDATIONS
To Builders and Users of Software

Make the most of effective software development technologies and formal methods. A variety of modern technologies — in particular, **safe programming languages**, **static analysis**, and **formal methods** — are likely to reduce the cost and difficulty of producing dependable software. Elementary best practices, such as source code control and systematic defect tracking, should be universally adopted, . . .

–Software for Dependable Systems

Patriot Missle

- **1991, Dhahran**
  - Iraqi Scud missile **hits barracks** killing 28 soldiers
  - Patriot system was activated, but **missed target** by over 600m
  - Floating point **representation of 0.1** was cause (rounding error over time, required 100 hours of continuous operation)
- **See:** “Engineering Disasters”
  `https://www.youtube.com/watch?v=EMVBLg2MrLs`

(Image by Darkone - Own work, CC BY-SA 2.5)

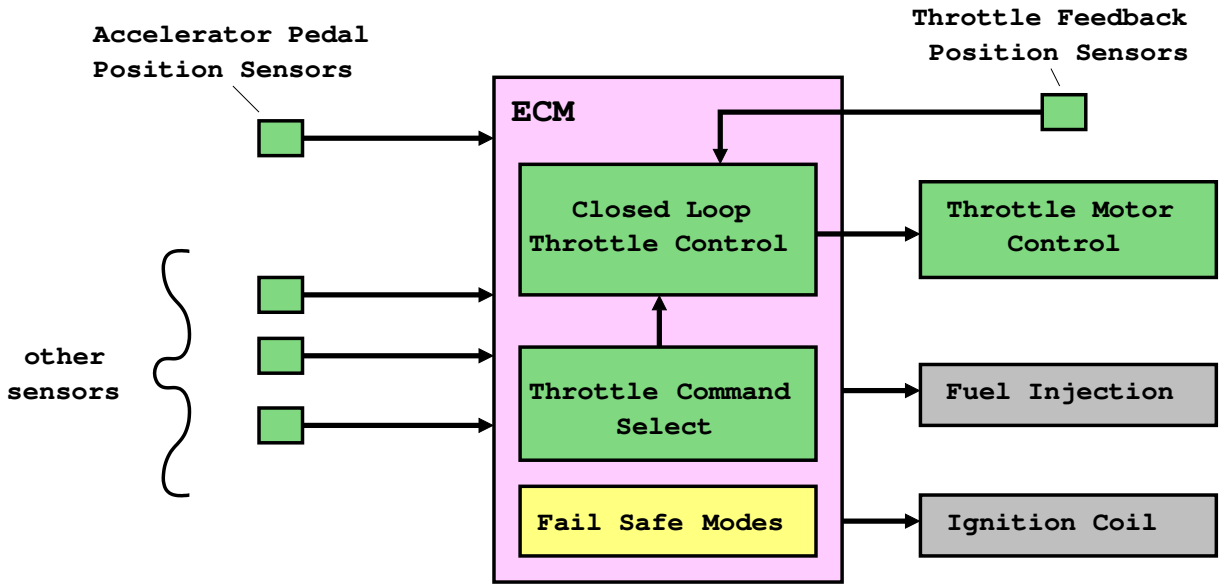
Unintended Acceleration? (Overview)

- Braking Problems (Toyota Prius)
  - Drivers reported **"sudden acceleration"** when braking
  - By 2010, over **seven million** cars recalled
  - **NASA experts** called into investigate
    Specialists in **static analysis**
    Their report had significant redactions though
  - Bookout vs Toyota, 2013
    Victim's awarded $1.5million each
    Barr provided **expert testimony**

See "Unintended Acceleration and Other Embedded Software bugs", Michael Barr, 2011

See "An Update on Toyota and Unintended acceleration"

Unintended Acceleration? (NASA Analysis)



"*The initial focus in analyzing the Camry MY 2005 source code has been on a thorough static source code analysis of the ECM source code to find possible coding defects and potential vulnerabilities in the code.*"

See "National Highway Traffic Safety Administration Toyota Unintended Acceleration Investigation"

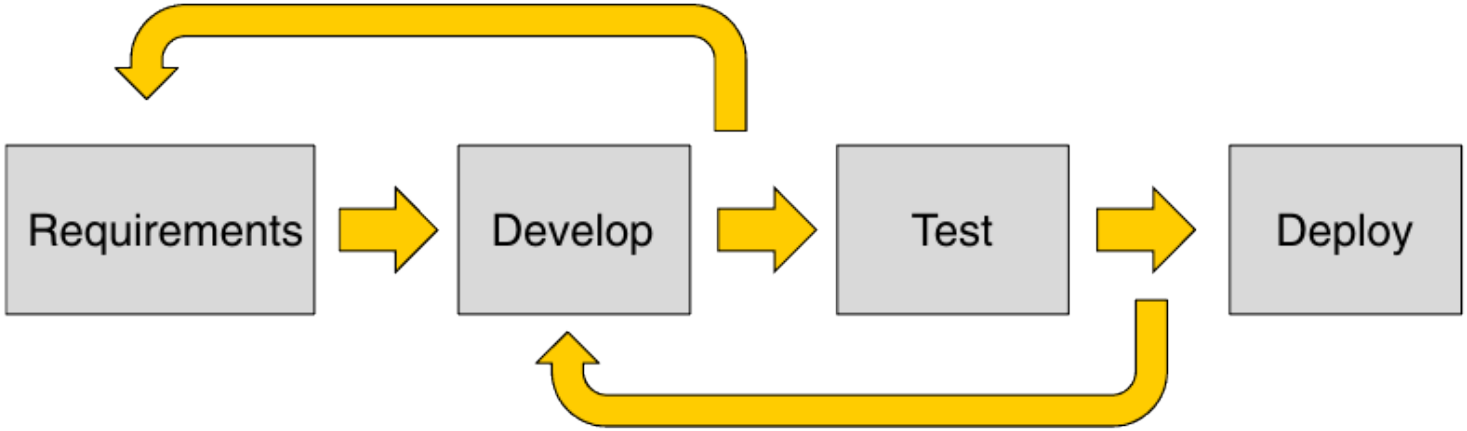
Unintended Acceleration? (Barr's Analysis)

"*Toyota's ETCS source code is of unreasonable quality. Toyota's source code is defective and contains bugs including bugs that can cause unintended acceleration*" –Michael Barr

- ECM main-cpu is V850E1 microcontroller (a 32-bit RISC CPU core)
- Code written in English following proprietary coding standard, sharing **similarities with MISRA-C**
- Redacted text indicates that compiler and static analysis tools produced large numbers of **warnings and errors**
- Code contained **243+ violations** of the "Power of 10-Rules"

See "An Update on Toyota and Unintended Acceleration", Michael Barr, 2013.

When to Catch Errors?



- Can we prevent errors in the **software requirements** themselves?
- Can we prevent errors being **introduced** during development?
- Can we prevent errors **through testing?**
- Is it **too late** to prevent errors at deployment time?

Example: *Type Checking*

```
/**
 * Type check an assertion statement. This requires checking that the
 * expression being asserted is well-formed and has boolean type.
 *
 * @param stmt
 *            Statement to type check
 * @param environment
 *            Determines the type of all variables immediately going into
 *            this block
 * @return
 */
private Environment propagate(Stmt.Assert stmt, Environment environment) {
    stmt.expr = propagate(stmt.expr, environment, current, null);
    checkIsSubtype(Type.T_BOOL, stmt.expr);
    return environment;
}
```

- **Aim:** Prevent type errors during development
- **Limitations:** Casting in Java means cannot catch all type errors
- **When:** During development, as *compile-time* or *static activity*

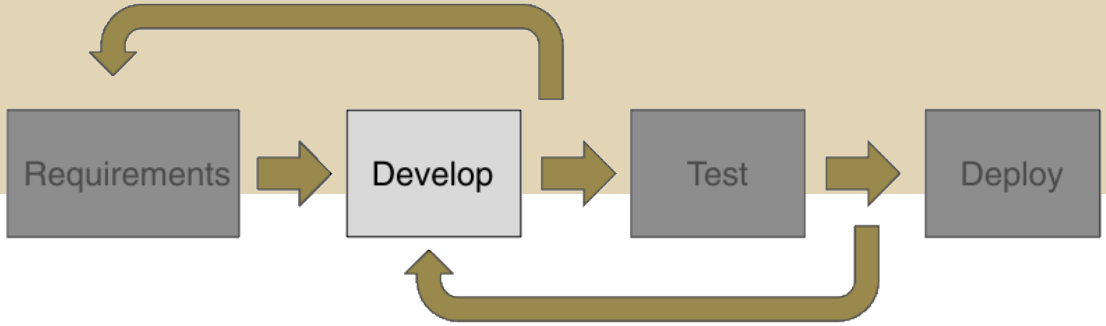
Type Checking is “Conservative”

```
public class Test {
    public static void fun(Test t) {
        // ...
    }

    public static void main(String[] args) {
        Object o = new Test();
        fun(o);
    }
}
```

- Type checking prohibits programs **with type errors**
- ... but also prohibits some programs **without type errors**
- ... and is also conservative (i.e. admits **false positives**)

“Compile Time”

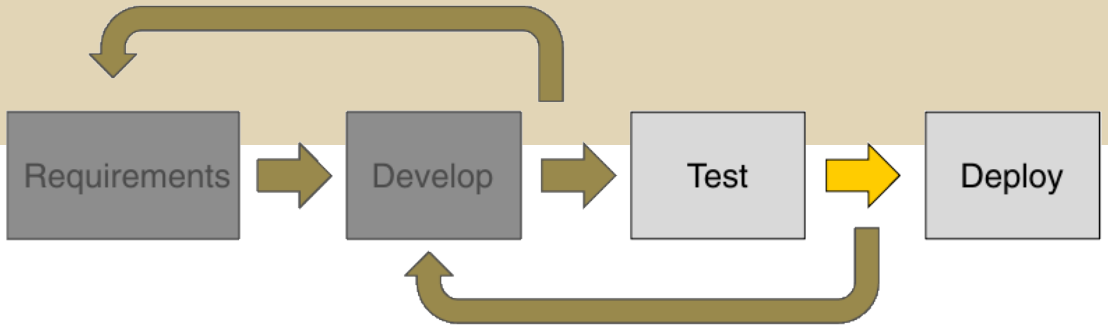


In computer science, **compile time** refers to either the operations performed by a compiler (the “compile-time operations”), programming language requirements that must be met by source code for it to be successfully compiled (the “compile-time requirements”), or properties of the program that can be reasoned about during compilation.

–Wikipedia

- Error Checking at Compile-Time
  - Occurs **before** program is run
  - Considers **all possible** execution traces
  - Possible to **guarantee** absence of certain errors
  - E.g. type checking in Java

“Run Time”



In computer science, **run time**, runtime or execution time is the time during which a program is running (executing), in contrast to other program lifecycle phases such as compile time, link time and load time.

–Wikipedia

- Error Checking at Runtime
  - Occurs during an **actual** program execution
  - Considers only **single** execution trace
  - Cannot guarantee absence of errors
  - E.g. runtime assertions in Java

Exercise: *Spot the Bug!*

```
// return largest of three numbers
int max3(int x, int y, int z) {
  int r;
  if(x > y) {
    if (x > z) { r = x; }
    else { r = z; }
  } else if (y > z) {
    r = z;
  } else {
    r = y;
  }
  //
  assert r >= x && r >= y && r >= z;
  return r;
}
```

- What are the **downsides** here?

Is it a Bug or a Feature?

A **functional specification** (also, functional spec, specs, functional specifications document (FSD), functional requirements specification) in systems engineering and software development is a document that specifies the functions that a system or component must perform (often part of a requirements specification) (ISO/IEC/IEEE 24765-2010).

–Wikipedia

```
int max(int[] items) {
  int max = items[0];
  for(int i=1;i<items.length;++i) {
    if(max < items[i]) { max = items[i]; }
  }
  return max;
}
```

- **Q)** is this function implemented **correctly?**