Software validation: Introduction

- Basic terminology
  - Static vs. dynamic validation techniques
- Code inspection
- Typing

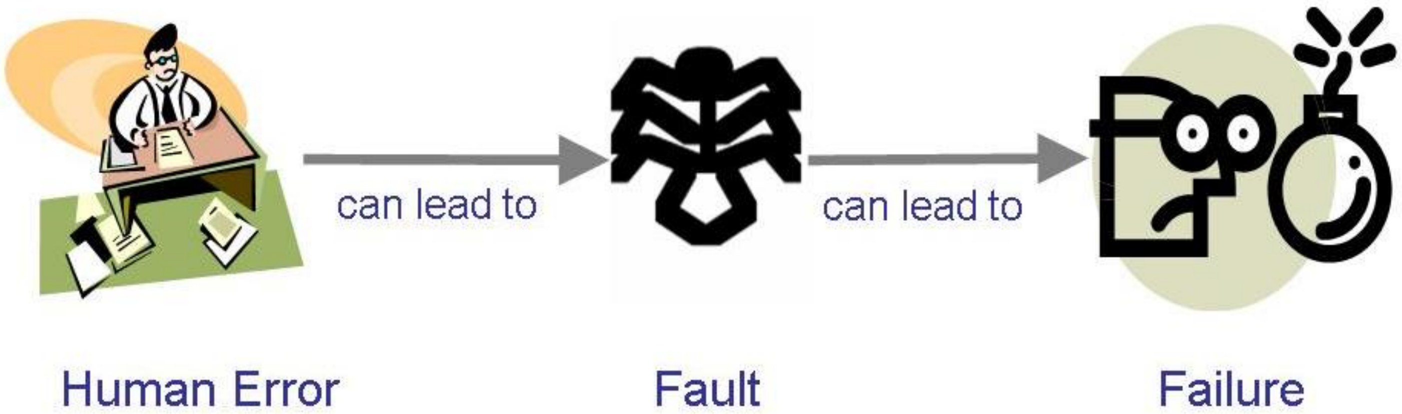
CE202 Software Engineering, Autumn term
Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex

Software verification and validation

- Checking that the implementation conforms to our expectations
  - Necessary condition: clear, unambiguous *requirement or product feature specification!*
- Two variations:
  - **Verification**: Checking the implementation wrt the <u>specification</u>
  - **Validation**: Checking the implementation wrt the <u>Client</u>
- Therefore:
  - *Validation* is what we "really" want
  - *Verification* is the 1st step towards *validation*

Terminology: error, fault & failure

- **Error**: A human mistake in SW development
  - May lead to one or more *faults*
- **Fault**: The result of error(s),
  - May lead to one or more failures
- **Failure:** The dynamic manifestation of fault(s)
  - Also: Departure from the required behaviour



Terminology: static vs. dynamic validation

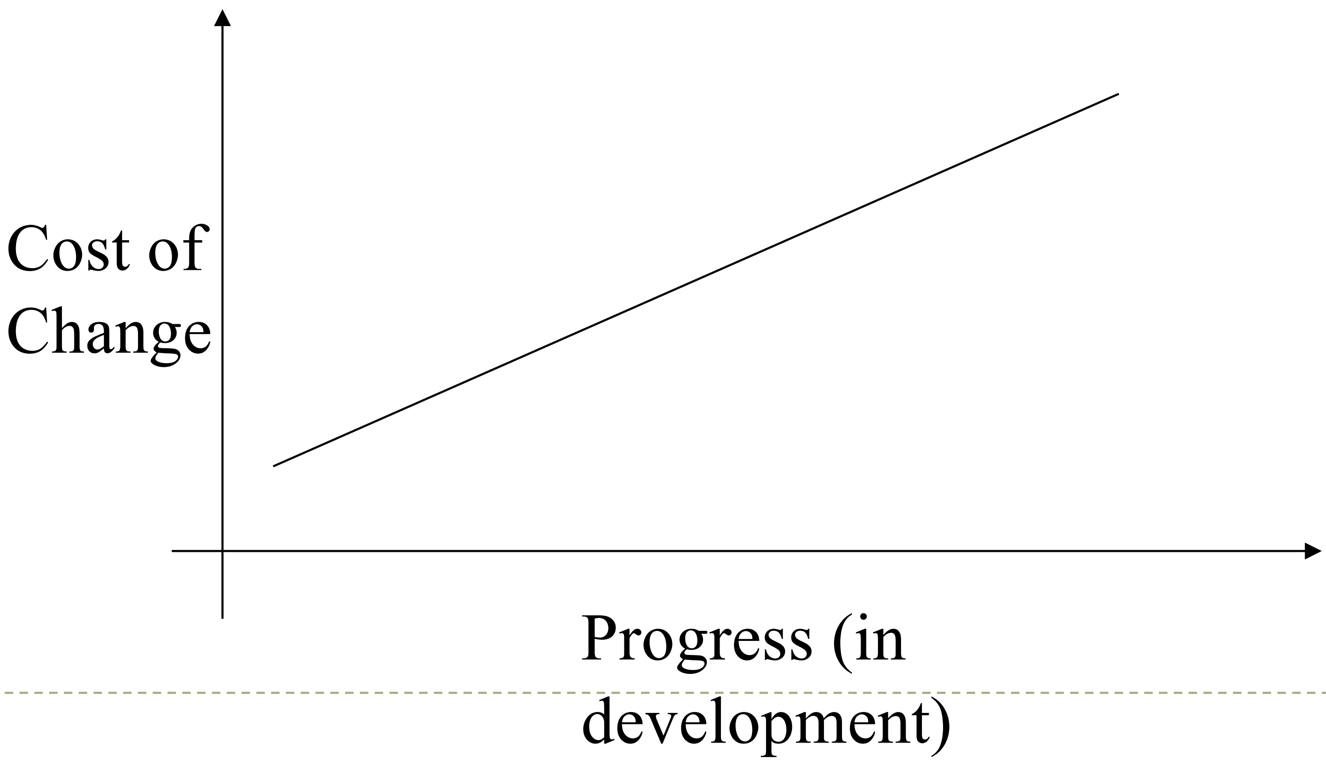
- **Static validation**: carried out without executing the program
  - Examples:
    - software inspection
    - formal verification
    - static typing
- **Dynamic validation**: carried out by executing the program
  - Examples:
    - Testing
    - Defensive programming (design by contract)

Static Vs. Dynamic Techniques

- Static validation does not require executing the program
- Pros:
  - Detection is earlier in the process
    - The system need not be complete
  - Correction is easier and cheaper
- Cons:
  - Reduced flexibility
  - Increased "bookkeeping"
  - Incomplete

Early Detection is Best

- Earlier changes are cheaper to make
  - Applies to both corrections and extensions



Software validation: code inspection

CE202 Software Engineering, Autumn term
Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex

Code Inspection

- AKA *program inspection*
- A static validation technique
- Informal review process carried by peers
  - Target: Source code
  - Resources required: Precise specification!
  - Objective: Detection of faults & anomalies
  - Local, step by step process
- Errors discovered:
  - Use of un-initialized data
  - Allocation and de-allocation of dynamic memory
  - Conditional statements that resolve statically
  - Exception checks
    - Appropriate handler for each exception
  - Array bounds
  - Infinite loops
  - Method not overridden
  - …

Tools supporting inspection

- Support a “checklist” of faults
- Examples: Java™ Compiler
  - Interface errors
  - typing errors (discussed separately)
- Example: C/C++ Lint
  - Variables declared but not used
  - Use of un-initialized variables
  - Unreachable code (“code coverage”)
  - Entry and exit points in loops
  - …

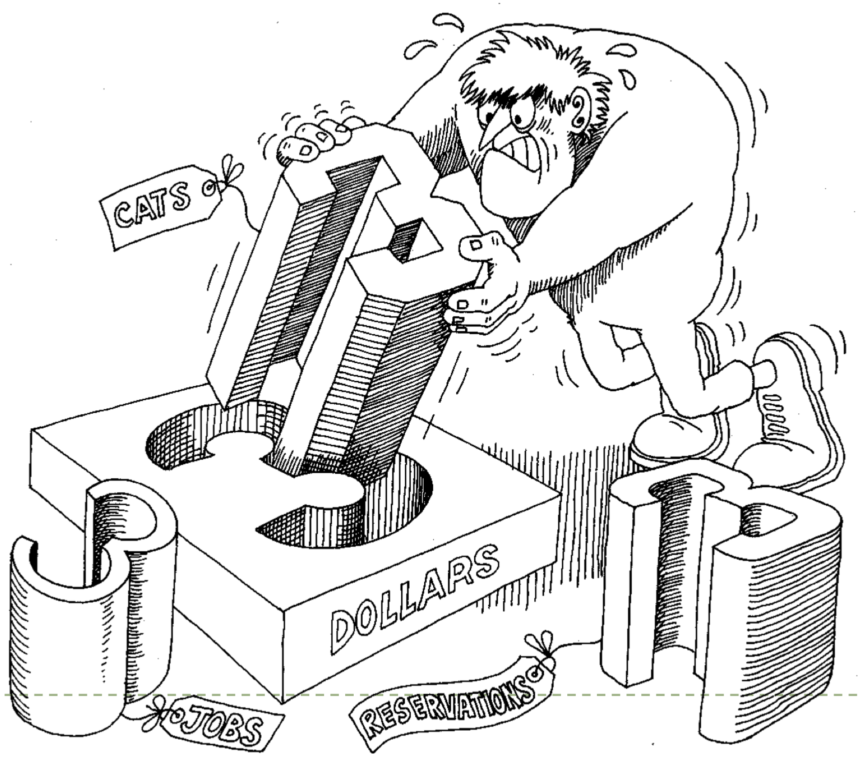
LINT Output Sample

```
138% more lint_ex.c
#include <stdio.h>
printarray (int Anarray) {
   printf(“%d”,Anarray);
}

main () {
  int Anarray[5]; int i; char c;
  printarray (Anarray, i, c);
  printarray (Anarray) ;
}

139% cc lint_ex.c
140% lint lint_ex.c

lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
```

Software validation: typing

Technique: Typing

- Supported by the programming language (e.g., Pascal, Ada, Java, C++ – partially)
  - Reduce the scope for errors
- Typing techniques:
  - Static/Dynamic typing
  - Strong/Weak typing
- Trade-offs
  - Safety Vs. Flexibility



Type

- A set of operations allowed over a group
- The interpretation of an area in memory
  - The sequence 0100 means different things if it is a signed integer, unsigned integer, an address, etc.

What is Typed?

- Anything that has a value
  - Variables
  - Expressions
  - Constants

Example: *Type* in C

- **`short`**
  - Representation: Two bytes
  - Operations: All integer operations
- **`short *`**
  - Representation: Address size
  - Operations: All pointer operations

Static Vs. Dynamic Typing

- ***Static typing***: *Type checking* performed statically
  - Done by the compiler
  - Requires type declaration for every name
  - Safer
  - Examples: All strongly-typed languages, C, C++, Eiffel, Java™
- ***Dynamic typing***: *Type checking* performed dynamically
  - Requires *dynamic binding* between the object and the type!
  - More flexible
  - Examples: Smalltalk, Lisp
  - Java: Cast operations are checked dynamically

Example: Static typing in Java

```
class IntStack {
  public void push(int) { ... }
  public int pop() {
      int result;
      ...
      return result;   // OK: result is of type int
  }
...
}
```

```
class UseStack {
  public String use(IntStack is, int k) {
      is.push(k);          // OK
      is.push("3");        // Typing error
      k.push(3);           // Typing error
      return "3";          // OK
      return 3;            // Typing error
  }
}
```

Static typing in Java (Cont.)

- In OOP: superclass is supertype

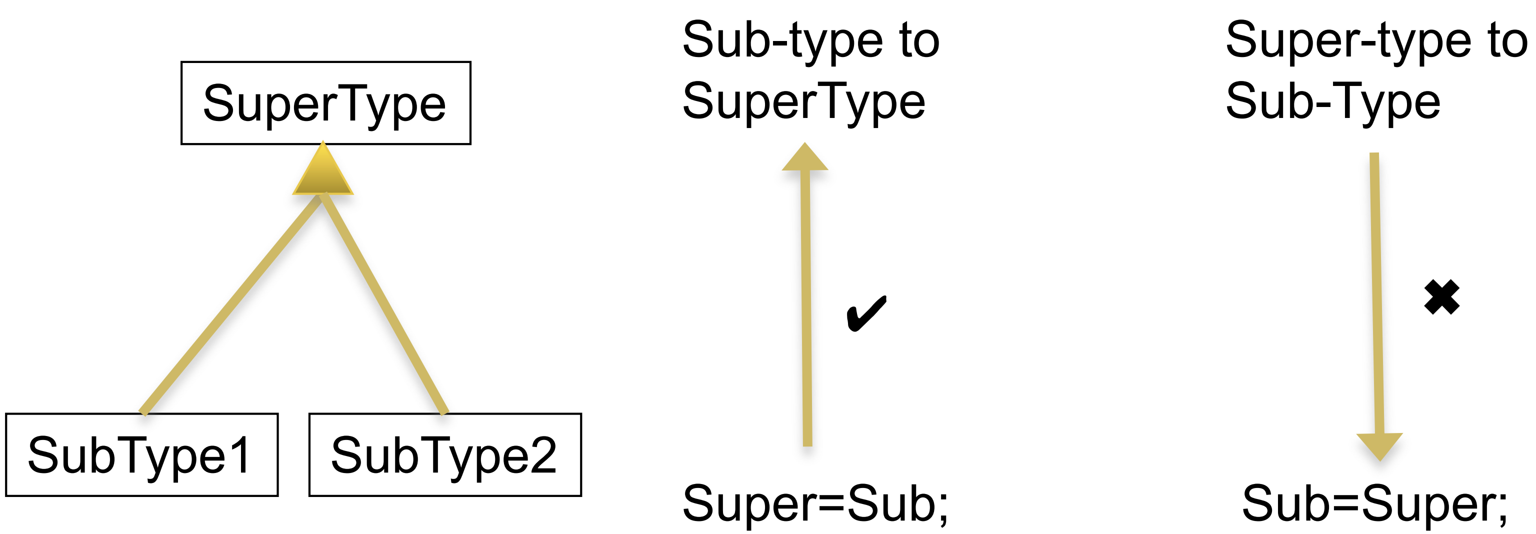
```
class UseStack {
  public string use(IntStack is) {
    int sz = is.pop();        // OK
    is.toString();            // OK
    Object obj = is;   // OK: Conversion to supertype
    obj.toString();    // OK
    obj.push(5);       // Typing error
    IntStack is2;
    is2 = obj;         // Error
    is2 = (IntStack)obj; // OK: Casting(will be checked dynamically)
String str=(String)obj; // OK: Casting(will fail in runtime)
}
}
```

Additional notes

- Remember ***obj*** is actually an instance of **InstStack**, as it is a pointer to the ***is*** object
- ***is*** is declared to be of class **IntStack**, so ***obj*** (which is actually an instance of **IntStack**) can be cast to **IntStack** dynamically
- Therefore, it is not possible for an instance of **IntStack** to be cast to a **String**.
- To understand this at a fundamental level, think about the ***is*** object as it would exist in the heap. It has a certain memory layout, with space allocated for the various fields and so on. ***is*** is a pointer to that memory structure, and then ***obj*** is created as another pointer to that structure. When you create **is2** as a pointer to that memory structure, the Java VM has to check your cast: so it ensures that the memory structure is type-compatible with an **IntStack** (which it is). However, when you create ***str*** as a pointer to that memory structure, the VM throws an exception (probably a ClassCastException?) to indicate that the memory structure is not compatible with an instance of a **String**.

Converting to the supertype

- It is permissible to convert from a sub-type to a super-type with automatic coercion
- It is not permissible to convert from a super-type to a sub-type without explicit casting



Type Coercion

- Relaxing *type compatibility:*
  - Allowed by most languages to different extents
- Two kinds of *type coercion*:
  - Implicit type conversions (AKA *standard conversion*): weakly typed languages, automatic type conversion by the compiler

```
int pi = 3.14159;
float x = '0';
unsigned int age = -1;
```

C

  - Explicit type conversion (AKA *cast*):

```
int pi = (int)3.14159;
```

C

float to **int** causes truncation, i.e. removal of the fractional part.
double to **float** causes rounding of digit
long int to **int** causes dropping of excess higher order bits.
`float x = '0'` => **x** is definitely a **float**, it is assigned to **48.00f** -- the float value of the character **'0'**

Coercion II

- What coercion is expected?

```
class LinkedList extends List {                    Java
   void method(List aList) {
      LinkedList aLinkedList = aList;        // Error: Requires cast
      LinkedList aLinkedList = (LinkedList)(aList);// OK, type checked
dynamically
      aList = aLinkedList;                       // OK (standard
conversion)
}
```

Strong Vs. Weak Typing

- **Weak typing:** support either implicit type, ad-hoc polymorphism (also known as overloading) or both.
- **Strong typing**: Coercions allowed only if value is preserved!
  - Other coercion operations: Cast (explicit coercion) is required
  - Strong typing implies static typing
- Pascal and Ada are strongly typed

```
type grade = 0..100;   // Subtype of integer   Ada
...
i : integer;
g : grade;
...
i := g;              // Coercion OK, value always preserved
g := i;              // Compilation error: Cast required
g := 1 + 1;          // Compilation error: Cast required
g := grade(i);       // Cast; Run-time checking will ensure that 0 ≤ i ≤ 100
g := grade(0);       //  Cast; Run-time checking will ensure that 0 is in range
```

Strong Vs. Weak Typing

	Weak Typing	Strong Typing
Pseudocode	a = 2 b = "2" concatenate(a, b) # Returns "22" add(a, b) # Returns 4	a = 2 b = "2" concatenate(a, b) # Type Error add(a, b) # Type Error concatenate(str(a), b) # Returns "22" add(a, int(b)) # Returns 4
Languages	BASIC, Perl, PHP, Rexx (is language dependent)	ActionScript 3, C++, C#, Java, Python, OCaml

Reminder: Static Vs. Dynamic Typing

- ***Static typing***: *Type checking* performed statically
  - Done by the compiler
  - Requires type declaration for every name
  - Safer
  - Examples: All strongly-typed languages, C, C++, Eiffel, Java™
- ***Dynamic typing***: *Type checking* performed dynamically
  - Requires *dynamic binding* between the object and the type!
  - More flexible
  - Examples: Smalltalk, Lisp
  - Java: Cast operations are checked dynamically

Example: dynamic typing in Smalltalk

- Smalltalk lacks static types
- Each object's class is known dynamically
- Type checks are only done dynamically

Smalltalk

```
StackUser>>use
      | aStack |                   "object type not known! "
      aStack := Stack new;         "object (dynamic) type is now set"
      aStack push;                 "OK: object's class recognizes message"
      aStack standOnYourHead;      "Failure: object's class does not recognize
                                   message"
"Method will compile successfully!"
```

Summary

- Software verification & validation - checking that the implementation conforms to our expectations
- The differences between Static vs Dynamic validation
- Validation through code inspection
- The use of Typing in programming languages
  - Static/Dynamic typing
  - Strong/Weak typing
  - Type coercion

Exercise: strong and weak typing

Which of the following is an example of a strong typing and which is an example of weak typing? Explain why?

```
/* Python code */
>>> foo = "x"
>>> foo = foo + 2
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in ?
    foo = foo + 2
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

Example 1

```
/* PHP code */
<?php
$foo = "x";
$foo = $foo + 2; // not an error
echo $foo;
?>
```

Example 2

Exercise: Strong and Weak Typing (2)

- What is good about dynamic typing?
- The following is an example from a dynamically typed language (Python). Why in this example, is dynamic typing a problem?

```
/* Python code */
my_variable = 10
while my_variable > 0:
        i = foo(my_variable)
        if i < 100:
                my_variable++
        else
                my_varaible = (my_variable + i) / 10 // spelling error intentional
```

End