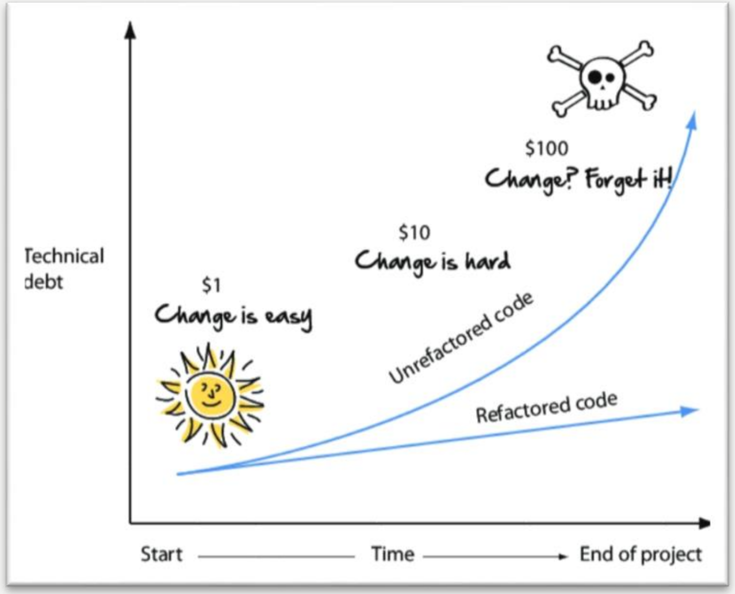
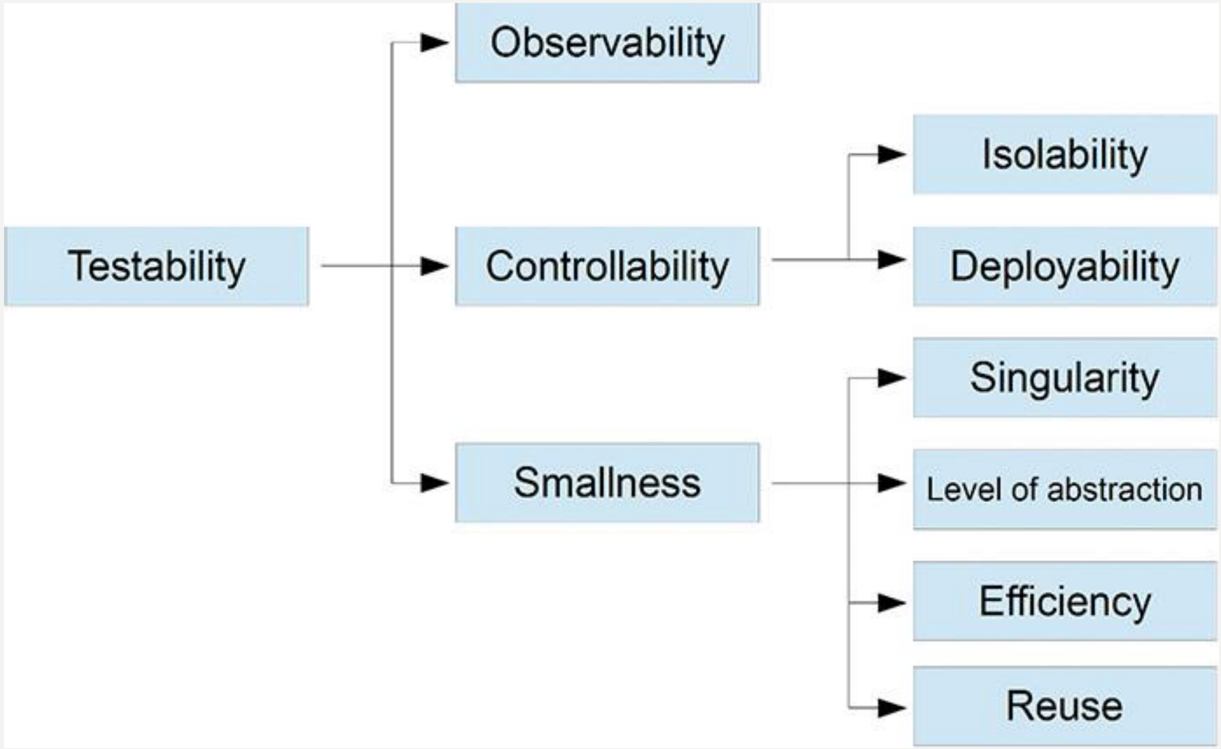
TESTABILITY



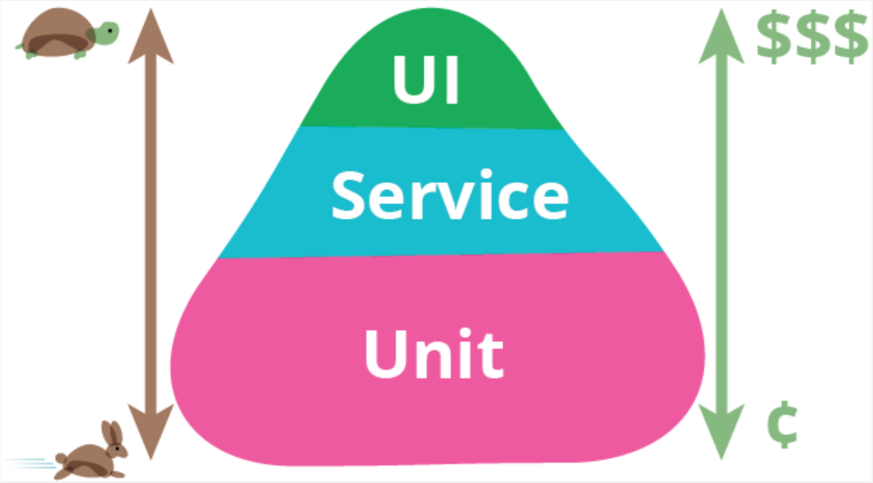
TESTABILITY CHARACTERISTICS

- **Observability**, **controllability** are cornerstones of testability.
- **Smallness** helps getting in that direction!



OBSERVABILITY

- **Logging** – too much and too little
- **Information hiding** – normally a good thing to separate interface from implementation
  - Tests at level beyond public API can become too coupled to internal representation
  - Shall I/ How to test private methods?



CONTROLLABILITY

- **Simple constructor** because we want to be able to:
  - Instantiate the class to test
  - Set the class into a particular state
  - Assert the final state of the class

- **Reduce dependencies** to be able to test program elements in isolation (easier to identify root cause)

REDUCE DEPENDENCIES– EXAMPLE

Every time we instantiate a `Vehicle` object, we also instantiate a `Driver` object

Problem: Application logic is mixed with instantiation code (factory code)

```
class Vehicle {
   Driver d = new Driver();

   boolean hasDriver = true;

   private void setHasDriver(boolean hasDriver)
   {
      this.hasDriver = hasDriver;
   }
}
```

REDUCE DEPENDENCIES– HOW

Solution: **Dependency injection**

Pass a **`Driver`** *interface* to the **`Vehicle`** class

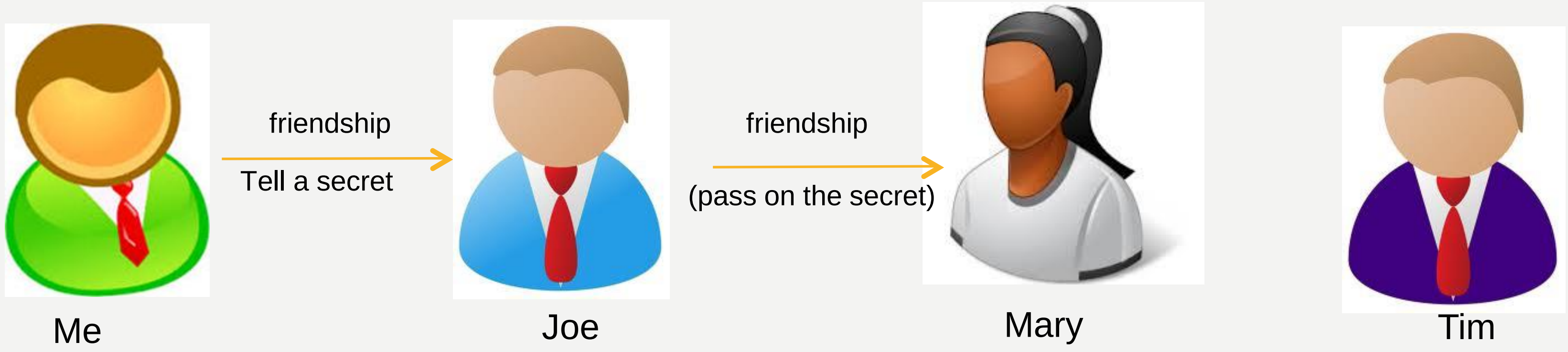
Separates application logic and instantiation logic → We can mock any type of **`Driver`** implementation

```
class Vehicle {
   Driver d;
   boolean hasDriver = true;

   Vehicle(Driver d) {
    this.d = d;
   }
   private void setHasDriver(boolean hasDriver) {
    this.hasDriver = hasDriver;
   }
}
```

AVOID HIDDEN DEPENDENCIES AND GLOBAL STATE

- Avoid global objects if they are not coded for shared access → they can give unintended consequences



Knows the secret – but how???

friendship

Tell a secret

friendship

(pass on the secret)

Me

Joe

Mary

Tim

Problem:
Only the one who originally built the relationships (code), knows the true dependencies!
To others, information can flow in some secret paths not clear to them ☹

AVOID HIDDEN DEPENDENCIES AND GLOBAL STATE

- Global state in action:
  - `DBManager` implies a global state.
  - Reservation object hides dependency upon a database manager

```
public void reserve() {
    DBManager manager = new DBManager();
    manager.initDatabase();
    Reservation r = new Reservation();
    r.reserve();
}
```

- Avoiding hidden dependency:

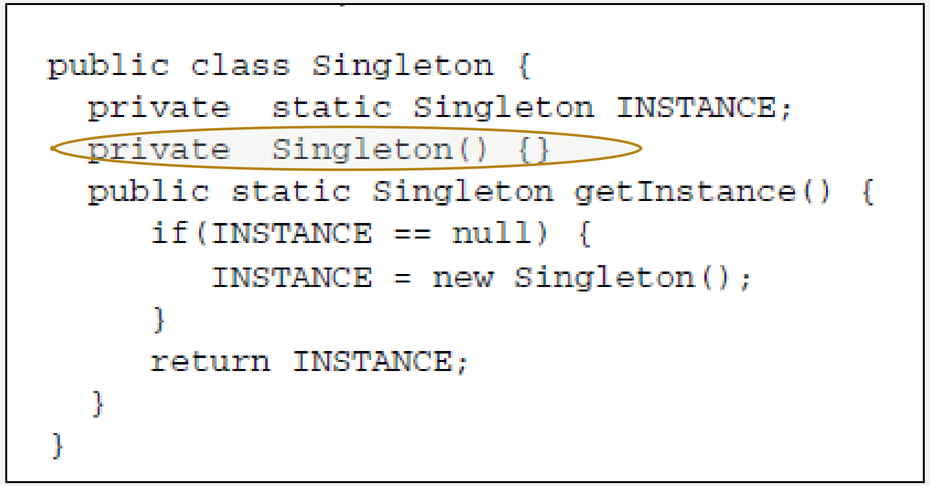
```
public void reserve() {
    DBManager manager = new DBManager();
    manager.initDatabase();
    Reservation r = new Reservation (manager);
    r.reserve();
}
```

Now clear that Reservation needs database manager

SINGLETONS PROS AND CONS

- Pro

    - object instantiated only once



```
public class Singleton {
  private  static Singleton INSTANCE;
  private  Singleton() {}
  public static Singleton getInstance() {
     if(INSTANCE == null) {
        INSTANCE = new Singleton();
     }
     return INSTANCE;
  }
}
```

- Cons
    - Can’t test a private method directly (constructor is private)
    - Solution
        - Rely on code coverage
        - Change access modifier while testing
        - Reflection
    - Introduces global state into code
        - when you provide access to a global object, you share not only that object but also any object to which it refers

CONTRACTS

- Public APIs are contracts

- Don’t just change the signature of public method!

Clients might break!

Tests can keep you on track

Be conservative in what you do, be liberal in what you accept from others

CONTRACTS AT METHOD LEVEL

- Preconditions
- Postconditions
- Variants

```
// pre: 0 < age
// post: returns true if age >= 18, otherwise false

public boolean legalAge(int age)
```

IMPOSSIBLE TO TEST EVERYTHING

```
int myMethod(int j) {
   j = j - 1; // should be j = j + 1
   j = j / 30000;
   return j;
}
```

input(j)	Expected output	Actual output
1	0	0
42	0	0
40000	1	1
-64000	-2	-2

Tests won’t find the bug

Example from *Testing Object-Oriented Systems* by Robert Binder

DOMAIN-TO-RANGE RATION PROBLEM

- Large input ranges with small output domains
- If ranges are (too) big, the bigger risk for not choosing the right test cases

Examples

- Odd and even numbers returning 0 and 1 respectively.

- Age range 0- 18 19 - ?

vs

- Age ranges 0- 18 19 – 26 27 – 75 76 - 120