

Know Your Units

TDD, DDT, POUTing and GUTs

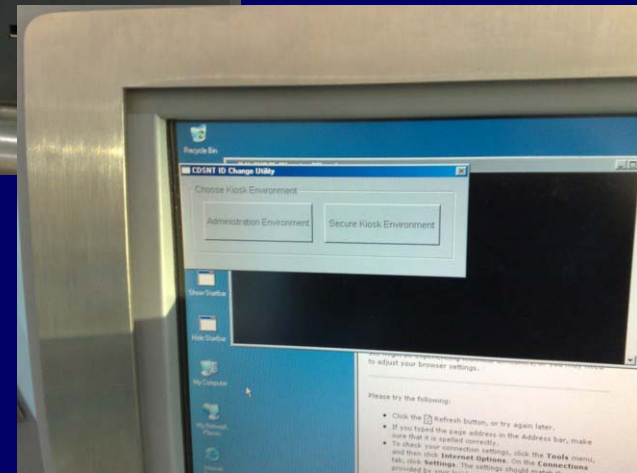
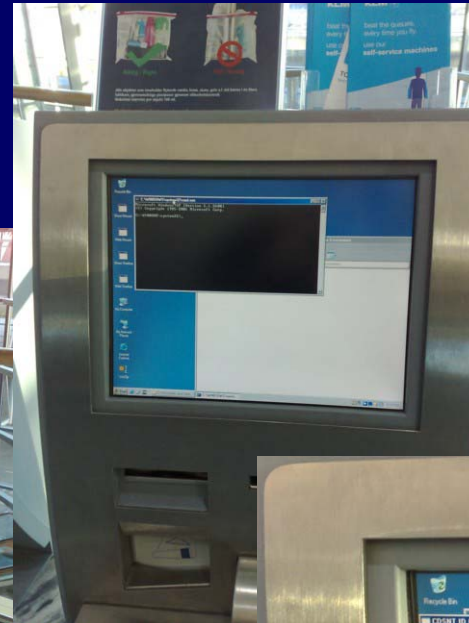
Kevlin Henney

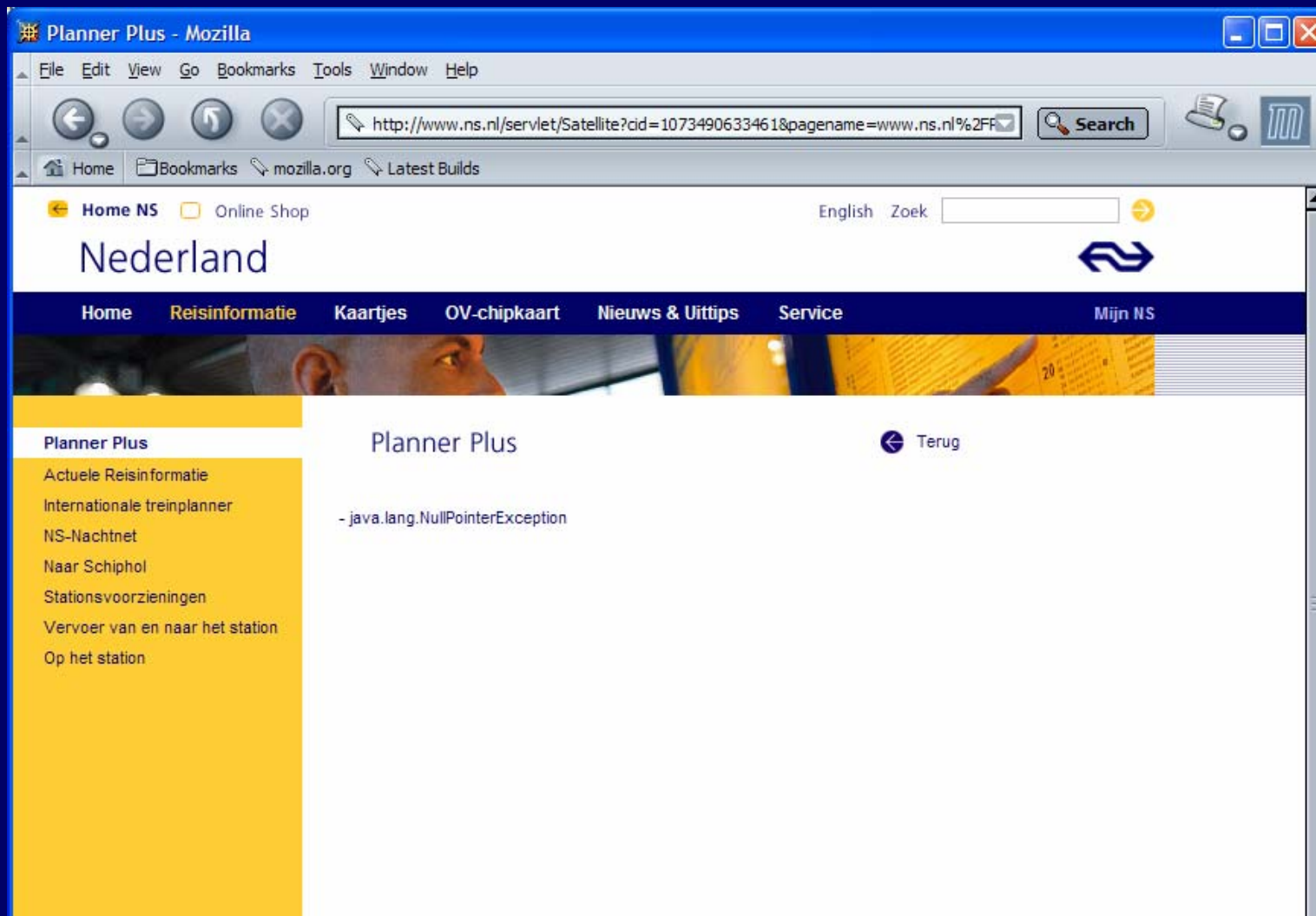
kevlin@curbralan.com

- **Intent**
 - Clarify the practice(s) of unit testing
- **Content**
 - Kinds of tests
 - Testing approach
 - Good unit tests
 - Listening to your tests
- **But first...**

```
Using autodetected IRQ (11) to improve performance.  
ifcusb (PC/TCP Class 1 packet driver - DIX Ethernet) init  
5 free packets of length 160, 5 free packets of length 1  
The kernel is using asynchronous sends  
The Resident Module occupies 0 bytes of conventional mem
```







Microsoft PowerPoint - [Pattern-Based Software Development (C#).ppt]

File Edit View Insert Format Tools Slide Show Window Help Adobe PDF

Type a question for help

Arial 14 B I U \$

72%

Design New Slide

Outline Slides

32 Typical Composite Design Patterns

33 Composite Data Structure in Java

34 The Factory Pattern

35 Factory Design Patterns

36 Patterns beyond OOP

37 Combining Patterns

38 Combining Patterns

Click to add notes

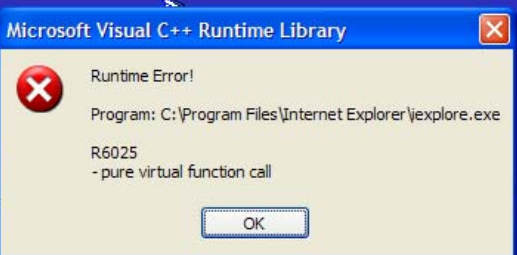
Draw AutoShapes

Slide 37 of 187

Template.ppt

English (U.K.)

Combining Patterns



Microsoft Visual C++ Runtime Library


Runtime Error!

Program: C:\Program Files\Internet Explorer\iexplore.exe

R6025

- pure virtual function call

OK



Pattern-Based Software Development

© Curbralan Ltd

37



Kinds of Tests

- **Testing is a way of showing that you care about something**
 - **If a particular quality has value for a system, there should be a concrete demonstration of that value**
 - **Code quality can be demonstrated through unit testing, static analysis, code reviews, analysing trends in defect and change history, etc.**

Unit testing involves testing the individual classes and mechanisms; is the responsibility of the application engineer who implemented the structure. [...]
System testing involves testing the system as a whole; is the responsibility of the quality assurance team.

Grady Booch

Object-Oriented Analysis and Design with Applications, 2nd edition

- **System testing involves testing of the software as a whole product**
 - **System testing may be a distinct job or role, but not necessarily a group**
- **Programmer testing involves testing of code by programmers**
 - **It is about code-facing tests, i.e., unit and integration tests**
 - **It is not the testing of programmers!**

Teams do deliver successfully using manual tests, so this can't be considered a *critical* success factor. However, every programmer I've interviewed who once moved to automated tests swore *never to work without them again*. I find this nothing short of astonishing.

Their reason has to do with improved quality of life. During the week, they revise sections of code knowing they can quickly check that they hadn't inadvertently broken something along the way. When they get code working on Friday, they go home knowing that they will be able on Monday to detect whether anyone had broken it over the weekend – they simply rerun the tests on Monday morning. The tests give them freedom of movement during the day and peace of mind at night.

Alistair Cockburn, *Crystal Clear*

- **Automated tests offer a way to reduce the waste of repetitive work**
 - **Write code that tests code**
- **However, note that not all kinds of tests can be automated**
 - **Usability tests require observation and monitoring of real usage**
 - **Exploratory testing is necessarily a manual task**

A test is not a unit test if:

- **It talks to the database**
- **It communicates across the network**
- **It touches the file system**
- **It can't run at the same time as any of your other unit tests**
- **You have to do special things to your environment (such as editing config files) to run it.**

Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness. However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes.

Michael Feathers, "A Set of Unit Testing Rules"

- **Unit tests are on code that is isolated from external dependencies**
 - **The outcome is based on the code of the test and the code under test**
- **Integration tests involve external dependencies**
 - **Some portion of a system is necessarily not unit testable; the portion that is not necessarily so is a measure of coupling**

- **It is worth distinguishing between tests on functional behaviour...**
 - **Written in terms of asserting values and interactions in response to use, the result of which is either right or wrong**
- **And tests on operational behaviour**
 - **More careful experimental design is needed because they typically require sampling and statistical interpretation**

Testing Approach

If all you could make was a long-term argument for testing, you could forget about it. Some people would do it out of a sense of duty or because someone was watching over their shoulder. As soon as the attention wavered or the pressure increased, no new tests would get written, the tests that were written wouldn't be run, and the whole thing would fall apart.

Kent Beck

Extreme Programming Explained, 1st edition

*Test Early.
Test Often.
Test Automatically.*

*Andrew Hunt and David Thomas
The Pragmatic Programmer*

- A *passive* approach looks at tests as a way of confirming code behaviour
 - This is Plain Ol' Unit Testing (POUTing)
- An *active* approach also uses testing to frame and review design
 - This is Test-Driven Development (TDD)
- A *reactive* approach introduces tests in response to defects
 - This is Defect-Driven Testing (DDT)

Very many people say "TDD" when they really mean, "I have good unit tests" ("I have GUTs"?) Ron Jeffries tried for years to explain what this was, but we never got a catch-phrase for it, and now TDD is being watered down to mean GUTs.

Alistair Cockburn

"The modern programming professional has GUTs"

- **TDD complements other design, verification and validation activities**
 - **The emphasis is on defining a contract for code with behavioural examples that act as both specification and confirmation**
 - **The act of writing tests (i.e., specifying) is interleaved with the act of writing the target code, offering both qualitative and quantitative feedback**

Because Unit Testing is the plain-Jane progenitor of Test Driven Development, it's kind of unfair that it doesn't have an acronym of its own. After all, it's hard to get programmer types to pay attention if they don't have some obscure jargon to bandy about. [...] I'll borrow from the telco folks and call unit testing Plain Old Unit Testing.

Jacob Proffitt, "TDD or POUT"

- **POUT employs testing as a code verification tool**
 - **Test writing follows code writing when the piece of target code is considered complete — effective POUTing assumes sooner rather than later**
 - **The emphasis is quantitative and the focus is on defect discovery rather than design quality or problem clarification**

- **DDT involves fixing a defect by first adding a test for the defect**
 - **DDT uses defects as an opportunity to improve coverage and embody learning from feedback in code form**
 - **This is a normal part of effective TDD and POUT practice**
 - **However, DDT can also be used on legacy code to grow the set of tests**

Less unit testing dogma.
More unit testing karma.

Alberto Savoia
"The Way of Testivus"

Good Unit Tests

I was using xUnit. I was using mock objects. So why did my tests seem more like necessary baggage instead of this glorious, enabling approach?

In order to resolve this mismatch, I decided to look more closely at what was inspiring all the unit-testing euphoria. As I dug deeper, I found some major flaws that had been fundamentally undermining my approach to unit testing.

The first realization that jumped out at me was that my view of testing was simply too “flat.” I looked at the unit-testing landscape and saw tools and technologies. The programmer in me made unit testing more about applying and exercising frameworks. It was as though I had essentially reduced my concept of unit testing to the basic mechanics of exercising xUnit to verify the behavior of my classes. [...]

In general, my mindset had me thinking far too narrowly about what it meant to write good unit tests.

Tod Golding, "Tapping into Testing Nirvana"

- **Poor unit tests lead to poor unit-testing experiences**
 - **Effective testing requires more than mastering the syntax of an assertion**
- **It is all too easy to end up with monolithic tests**
 - **Rambling stream-of-consciousness narratives intended for machine execution but not human consumption**

```

[Test]
public void Test()
{
    RecentlyUsedList list = new RecentlyUsedList();
    Assert.AreEqual(0, list.Count);
    list.Add("Aardvark");
    Assert.AreEqual(1, list.Count);
    Assert.AreEqual("Aardvark", list[0]);
    list.Add("Zebra");
    list.Add("Mongoose");
    Assert.AreEqual(3, list.Count);
    Assert.AreEqual("Mongoose", list[0]);
    Assert.AreEqual("Zebra", list[1]);
    Assert.AreEqual("Aardvark", list[2]);
    list.Add("Aardvark");
    Assert.AreEqual(3, list.Count);
    Assert.AreEqual("Aardvark", list[0]);
    Assert.AreEqual("Mongoose", list[1]);
    Assert.AreEqual("Zebra", list[2]);
    bool thrown;
    try
    {
        string unreachable = list[3];
        thrown = false;
    }
    catch (ArgumentOutOfRangeException)
    {
        thrown = true;
    }
    Assert.IsTrue(thrown);
}

```

```

[Test]
public void Test1()
{
    RecentlyUsedList list = new RecentlyUsedList();
    Assert.AreEqual(0, list.Count);
    list.Add("Aardvark");
    Assert.AreEqual(1, list.Count);
    Assert.AreEqual("Aardvark", list[0]);
    list.Add("Zebra");
    list.Add("Mongoose");
    Assert.AreEqual(3, list.Count);
    Assert.AreEqual("Mongoose", list[0]);
    Assert.AreEqual("Zebra", list[1]);
    Assert.AreEqual("Aardvark", list[2]);
}

[Test]
public void Test2()
{
    RecentlyUsedList list = new RecentlyUsedList();
    Assert.AreEqual(0, list.Count);
    list.Add("Aardvark");
    Assert.AreEqual(1, list.Count);
    Assert.AreEqual("Aardvark", list[0]);
    list.Add("Zebra");
    list.Add("Mongoose");
    Assert.AreEqual(3, list.Count);
    Assert.AreEqual("Mongoose", list[0]);
    Assert.AreEqual("Zebra", list[1]);
    Assert.AreEqual("Aardvark", list[2]);
    list.Add("Aardvark");
    Assert.AreEqual(3, list.Count);
    Assert.AreEqual("Aardvark", list[0]);
    Assert.AreEqual("Mongoose", list[1]);
    Assert.AreEqual("Zebra", list[2]);
}

[Test]
public void Test3()
{
    RecentlyUsedList list = new RecentlyUsedList();
    Assert.AreEqual(0, list.Count);
    list.Add("Aardvark");

```

- **A predominantly procedural test style is rarely effective**
 - **It is based on the notion that "I have a function *foo*, therefore I have one test function that tests *foo*", which does not really make sense for object usage**
 - **And, in truth, procedural testing has never really made sense for procedural code either**

```

[Test]
public void Constructor()
{
    RecentlyUsedList list = new RecentlyUsedList();
    Assert.AreEqual(0, list.Count);
}
[Test]
public void Add()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.Add("Aardvark");
    Assert.AreEqual(1, list.Count);
    list.Add("Zebra");
    list.Add("Mongoose");
    Assert.AreEqual(3, list.Count);
    list.Add("Aardvark");
    Assert.AreEqual(3, list.Count);
}
[Test]
public void Indexer()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.Add("Aardvark");
    list.Add("Zebra");
    list.Add("Mongoose");
    Assert.AreEqual("Mongoose", list[0]);
    Assert.AreEqual("Zebra", list[1]);
    Assert.AreEqual("Aardvark", list[2]);
    list.Add("Aardvark");
    Assert.AreEqual("Aardvark", list[0]);
    Assert.AreEqual("Mongoose", list[1]);
    Assert.AreEqual("Zebra", list[2]);
    bool thrown;
    try
    {
        string unreachable = list[3];
        thrown = false;
    }
    catch (ArgumentOutOfRangeException)
    {
        thrown = true;
    }
    Assert.IsTrue(thrown);
}

```

Constructor

Add

Indexer


```

void test_sort()
{
    int single[] = { 2 };
    quickersort(single, 1);
    assert(sorted(single, 1));

    int identical[] = { 2, 2, 2 };
    quickersort(identical, 3);
    assert(sorted(identical, 3));

    int ascending[] = { -1, 0, 1 };
    quickersort(ascending, 3);
    assert(sorted(ascending, 3));

    int descending[] = { 1, 0, -1 };
    quickersort(descending, 3);
    assert(sorted(descending, 3));

    int arbitrary[] = { 32, 12, 19, INT_MIN };
    quickersort(arbitrary, 4);
    assert(sorted(arbitrary, 4));
}

```

```

void test_sort()
{
    int empty[] = { 2, 1 };
    quickersort(empty + 1, 0);
    assert(empty[0] == 2);
    assert(empty[1] == 1);

    int single[] = { 3, 2, 1 };
    quickersort(single + 1, 1);
    assert(single[0] == 3);
    assert(single[1] == 2);
    assert(single[2] == 1);

    int identical[] = { 3, 2, 2, 2, 1 };
    quickersort(identical + 1, 3);
    assert(identical[0] == 3);
    assert(identical[1] == 2);
    assert(identical[2] == 2);
    assert(identical[3] == 2);
    assert(identical[4] == 1);

    int ascending[] = { 2, -1, 0, 1, -2 };
    quickersort(ascending + 1, 3);
    assert(ascending[0] == 2);
    assert(ascending[1] == -1);
    assert(ascending[2] == 0);
    assert(ascending[3] == 1);
    assert(ascending[4] == -2);

    int descending[] = { 2, 1, 0, -1, -2 };
    quickersort(descending + 1, 3);
    assert(descending[0] == 2);
    assert(descending[1] == -1);
    assert(descending[2] == 0);
    assert(descending[3] == 1);
    assert(descending[4] == -2);

    int arbitrary[] = { 100, 32, 12, 19, INT_MIN, 0 };
    quickersort(arbitrary + 1, 4);
    assert(arbitrary[0] == 100);
    assert(arbitrary[1] == INT_MIN);
    assert(arbitrary[2] == 12);
    assert(arbitrary[3] == 19);
    assert(arbitrary[4] == 32);
    assert(arbitrary[5] == 0);
}

```

Everybody knows that TDD stands for Test Driven Development. However, people too often concentrate on the words "Test" and "Development" and don't consider what the word "Driven" really implies. For tests to drive development they must do more than just test that code performs its required functionality: they must clearly express that required functionality to the reader. That is, they must be clear specifications of the required functionality. Tests that are not written with their role as specifications in mind can be very confusing to read. The difficulty in understanding what they are testing can greatly reduce the velocity at which a codebase can be changed.

**Nat Pryce and Steve Freeman
"Are Your Tests Really Driving Your Development?"**

- **Behavioural tests are based on usage scenarios and outcomes**
 - **They are purposeful, cutting across and combine individual operations and use**
 - **Test names should reflect requirements, emphasising intention and goals rather than mechanics**
 - **This style correlates with the idea of use cases and user stories in the small**

```

[Test]
public void InitialListIsEmpty()
{
    RecentlyUsedList list = new RecentlyUsedList();
    Assert.AreEqual(0, list.Count);
}
[Test]
public void AdditionOfSingleItemToEmptyListIsRetained()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.Add("Aardvark");

    Assert.AreEqual(1, list.Count);
    Assert.AreEqual("Aardvark", list[0]);
}
[Test]
public void AdditionOfDistinctItemsIsRetainedInStackOrder()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.Add("Aardvark");
    list.Add("Zebra");
    list.Add("Mongoose");

    Assert.AreEqual(3, list.Count);
    Assert.AreEqual("Mongoose", list[0]);
    Assert.AreEqual("Zebra", list[1]);
    Assert.AreEqual("Aardvark", list[2]);
}
[Test]
public void DuplicateItemsAreMovedToFrontButNotAdded()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.Add("Aardvark");
    list.Add("Mongoose");
    list.Add("Aardvark");

    Assert.AreEqual(2, list.Count);
    Assert.AreEqual("Aardvark", list[0]);
    Assert.AreEqual("Mongoose", list[1]);
}
[Test, ExpectedException(ExceptionType = typeof(ArgumentOutOfRangeException))]
public void OutOfRangeIndexThrowsException()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.Add("Aardvark");
    list.Add("Mongoose");
    list.Add("Aardvark");
    string unreachable = list[3];
}

```

Initial list is empty

Addition of single item to empty list is retained

Addition of distinct items is retained in stack order

Duplicate items are moved to front but not added

Out of range index throws exception

```

void require_that_sorting_nothing_does_not_overrun()
{
    int empty[] = { 2, 1 };
    quickersort(empty + 1, 0);
    assert(empty[0] == 2);
    assert(empty[1] == 1);
}

void require_that_sorting_single_value_changes_nothing()
{
    int single[] = { 3, 2, 1 };
    quickersort(single + 1, 1);
    assert(single[0] == 3);
    assert(single[1] == 2);
    assert(single[2] == 1);
}

void require_that_sorting_identical_value_changes_nothing()
{
    int identical[] = { 3, 2, 2, 2, 1 };
    quickersort(identical + 1, 3);
    assert(identical[0] == 3);
    assert(identical[1] == 2);
    assert(identical[2] == 2);
    assert(identical[3] == 2);
    assert(identical[4] == 1);
}

void require_that_sorting_ascending_sequence_changes_nothing()
{
    int ascending[] = { 2, -1, 0, 1, -2 };
    quickersort(ascending + 1, 3);
    assert(ascending[0] == 2);
    assert(ascending[1] == -1);
    assert(ascending[2] == 0);
    assert(ascending[3] == 1);
    assert(ascending[4] == -2);
}

void require_that_sorting_descending_sequence_reverses_it()
{
    int descending[] = { 2, 1, 0, -1, -2 };
    quickersort(descending + 1, 3);
    assert(descending[0] == 2);
    assert(descending[1] == -1);
    assert(descending[2] == 0);
    assert(descending[3] == 1);
    assert(descending[4] == -2);
}

void require_that_sorting_mixed_sequence_orders_it()
{
    int arbitrary[] = { 100, 32, 12, 19, INT_MIN, 0 };
    quickersort(arbitrary + 1, 4);
    assert(arbitrary[0] == 100);
    assert(arbitrary[1] == INT_MIN);
    assert(arbitrary[2] == 12);
    assert(arbitrary[3] == 19);
    assert(arbitrary[4] == 32);
    assert(arbitrary[5] == 0);
}

```

Require that sorting nothing does not overrun

Require that sorting single value changes nothing

Require that sorting identical values changes nothing

Require that sorting ascending sequence changes nothing

Require that sorting descending sequence reverses it

Require that sorting mixed sequence orders it

- **Example-based test cases are easy to read and simple to write**
 - **They have a linear narrative with a low cyclomatic complexity, and are therefore less tedious and error prone**
 - **Coverage of critical cases is explicit**
 - **Additional value coverage can be obtained using complementary and more exhaustive, data-driven tests**



- **There are a number of things that should appear in tests**
 - **Simple cases, because you have to start somewhere**
 - **Common cases, using equivalence partitioning to identify representatives**
 - **Boundary cases, testing at and around**
 - **Contractual error cases, i.e., test rainy-day as well as happy-day scenarios**

- **There are a number of things that should not appear in tests**
 - **Do not assert on behaviours that are standard for the platform or language — the tests should be on your code**
 - **Do not assert on implementation specifics — a comparison may return *1* but test for > 0 — or incidental presentation — spacing, spelling, etc.**

Refactoring (noun): *a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.*

Refactor (verb): *to restructure software by applying a series of refactorings without changing the observable behavior of the software.*

Martin Fowler, *Refactoring*

```

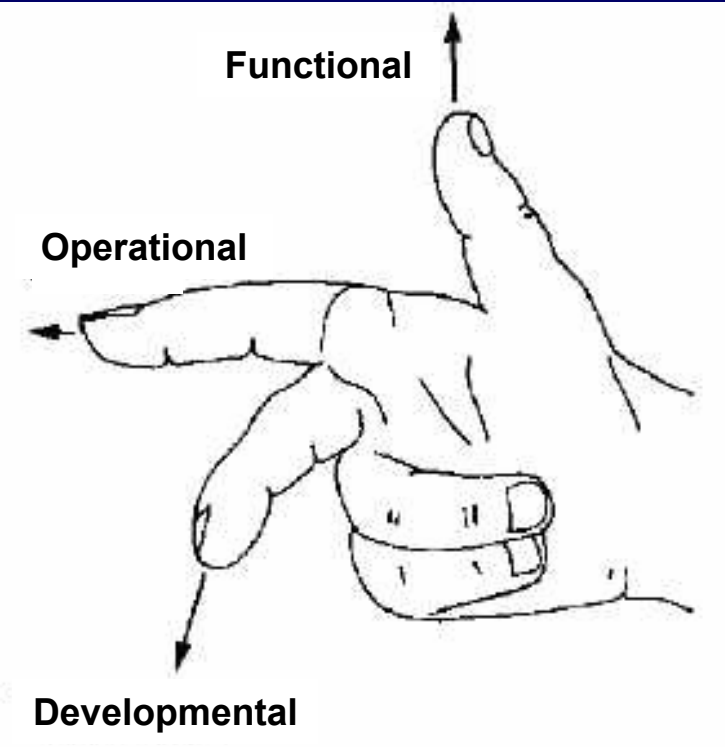
public class RecentlyUsedList
{
    public RecentlyUsedList()
    {
        list = new List<string>();
    }
    public void Add(string newItem)
    {
        if (list.Contains(newItem))
        {
            int position = list.IndexOf(newItem);
            string existingItem = list[position];
            list.RemoveAt(position);
            list.Insert(0, existingItem);
        }
        else
        {
            list.Insert(0, newItem);
        }
    }
    public int Count
    {
        get
        {
            int size = list.Count;
            return size;
        }
    }
    public string this[int index]
    {
        get
        {
            int position = 0;
            foreach (string value in list)
            {
                if (position == index)
                    return value;
                ++position;
            }
            throw new ArgumentOutOfRangeException();
        }
    }
    private List<string> list;
}

```

```

public class RecentlyUsedList
{
    public void Add(string newItem)
    {
        list.Remove(newItem);
        list.Add(newItem);
    }
    public int Count
    {
        get
        {
            return list.Count;
        }
    }
    public string this[int index]
    {
        get
        {
            return list[Count - index - 1];
        }
    }
    private List<string> list = new List<string>();
}

```



```

class access_control
{
public:
    bool is_locked(const std::basic_string<char> &key) const
    {
        std::list<std::basic_string<char> >::const_iterator found = std::find(locked.begin(), locked.end(), key);
        return found != locked.end();
    }
    bool lock(const std::basic_string<char> &key)
    {
        std::list<std::basic_string<char> >::iterator found = std::find(locked.begin(), locked.end(), key);
        if(found == locked.end())
        {
            locked.insert(locked.end(), key);
            return true;
        }
        return false;
    }
    bool unlock(const std::basic_string<char> &key)
    {
        std::list<std::basic_string<char> >::iterator found = std::find(locked.begin(), locked.end(), key);
        if(found != locked.end())
        {
            locked.erase(found);
            return true;
        }
        return false;
    }
    ...
private:
    std::list<std::basic_string<char> > locked;
    ...
};

```

```
class access_control
{
public:
    bool is_locked(const std::string &key) const
    {
        return std::count(locked.begin(), locked.end(), key) != 0;
    }
    bool lock(const std::string &key)
    {
        if(is_locked(key))
        {
            return false;
        }
        else
        {
            locked.push_back(key);
            return true;
        }
    }
    bool unlock(const std::string &key)
    {
        const std::size_t old_size = locked.size();
        locked.remove(key);
        return locked.size() != old_size;
    }
    ...
private:
    std::list<std::string> locked;
    ...
};
```

```
class access_control
{
public:
    bool is_locked(const std::string &key) const
    {
        return locked.count(key) != 0;
    }
    bool lock(const std::string &key)
    {
        return locked.insert(key).second;
    }
    bool unlock(const std::string &key)
    {
        return locked.erase(key);
    }
    ...
private:
    std::set<std::string> locked;
    ...
};
```

- **Ideally, unit tests are black-box tests**
 - They should focus on interface contract
 - They should not touch object internals or assume control structure in functions
- **White-box tests can end up testing that the code does what it does**
 - They sometimes end up silencing opportunities for refactoring, as well as undermine attempts at refactoring

Listening to Your Tests

- **Tests and testability offer many forms of feedback on code quality**
 - **But you need to take care to listen (not just hear) and understand what it means and how to respond to it**
 - **Don't solve symptoms — "Unit testing is boring/difficult/impossible, therefore don't do unit testing" — identify and resolve root causes — "Why is unit testing boring/difficult/impossible?"**

- **Unit testability is a property of loosely coupled code**
 - **Inability to unit test a significant portion of the code base is an indication of dependency problems**
 - **If only a small portion of the code is unit testable, it is likely that unit testing will not appear to be pulling its weight in contrast to, say, integration testing**



- **Unit testing also offers feedback on code cohesion, and vice versa**
 - **Overly complex behavioural objects that are essentially large slabs of procedural code**
 - **Anaemic, underachieving objects that are plain ol' data structures in disguise**
 - **Objects that are incompletely constructed in need of validation**



```

[Test]
public void AdditionOfSingleItemToEmptyListIsRetained()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.List = new List<string>();
    list.Add("Aardvark");

    Assert.AreEqual(1, list.List.Count);
    Assert.AreEqual("Aardvark", list.List[0]);
}
[Test]
public void AdditionOfDistinctItemsIsRetainedInStackOrder()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.List = new List<string>();
    list.Add("Aardvark");
    list.Add("Zebra");
    list.Add("Mongoose");

    Assert.AreEqual(3, list.List.Count);
    Assert.AreEqual("Mongoose", list.List[0]);
    Assert.AreEqual("Zebra", list.List[1]);
    Assert.AreEqual("Aardvark", list.List[2]);
}
[Test]
public void DuplicateItemsAreMovedToFrontButNotAdded()
{
    RecentlyUsedList list = new RecentlyUsedList();
    list.List = new List<string>();
    list.Add("Aardvark");
    list.Add("Mongoose");
    list.Add("Aardvark");

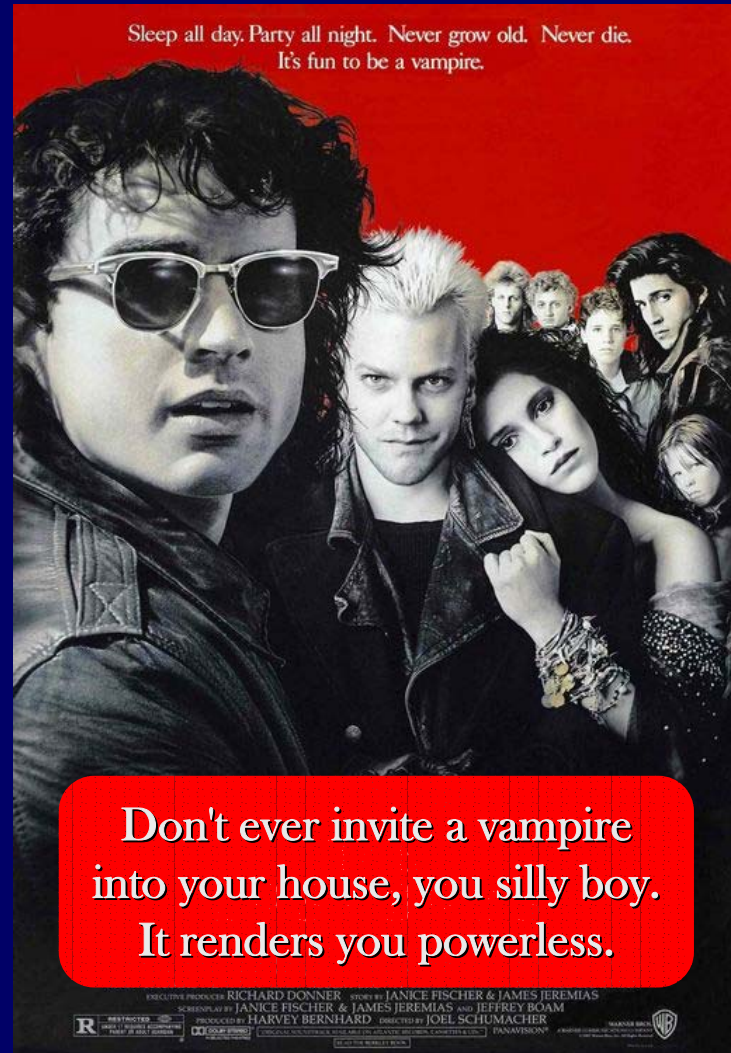
    Assert.AreEqual(2, list.List.Count);
    Assert.AreEqual("Aardvark", list.List[0]);
    Assert.AreEqual("Mongoose", list.List[1]);
}

```

```

public class RecentlyUsedList
{
    public void Add(string newItem)
    {
        list.Remove(newItem);
        list.Insert(0, newItem);
    }
    public List<string> List
    {
        get
        {
            return list;
        }
        set
        {
            list = value;
        }
    }
    private List<string> list;
}

```



- **Be careful with coverage metrics**
 - Low coverage is always a warning sign
 - Otherwise, understanding what kind of coverage is being discussed is key — is coverage according to statement, condition, path or value?
- **And be careful with coverage myths**
 - Without a context of execution, plain assertions in code offer *no* coverage

- **Watch out for misattribution**
 - **If you have GUTs, but you fail to meet system requirements, the problem lies with system testing and managing requirements, not with unit testing**
 - **If you don't know what to test, then how do you know what to code? Not knowing *how* to test is a separate issue addressed by learning and practice**

- **Don't shoot the messenger**
 - **If the bulk of testing occurs late in development, it is likely that unit testing will be difficult and will offer a poor ROI**
 - **If TDD or pre-check-in POUTing is adopted for new code, but fewer defects are logged than with late testing on the old code, that is (1) good news and (2) unsurprising**

