Dynamic Typing and Static Analysis

BY EVAN OLDS

CPT S 422

Example Code

- Consider these 2 C# code examples:

```
var v = 36;
v += 6;
Console.WriteLine(v);
```

```
dynamic d = 36;
d += 6;
Console.WriteLine(d);
```

- Both WriteLine statements produce an output of ‘42’

First Question

- First logical question should be:
  - What's the difference between "dynamic" and "var"?
- But before we get to that
  - Experiment with code
  - Make alterations to the code and see what the compiler and/or runtime results tell you about what's going on
- For the previous example
  - Initialized with integer value
  - Added integer value
  - What happens if we initialize as integer and try to add string?

Example Code Altered

- What is the output of each app? Do they even compile?

```
public static void Main(string[] args)
{
    var v = 36;
    v += "Hello";
    Console.WriteLine(v);
}

public static void Main(string[] args)
{
    dynamic d = 36;
    d += "Hello";
    Console.WriteLine(d);
}
```

First App Doesn't Compile

- Recall purpose of '**var**' keyword:
  - Compiler determines type automatically for you
  - Therefore, type is determined to be **int** from the declaration/initialization line
  - Can't concatenate string onto **int**

```
var v = 36;
v += "Hello";
Console.WriteLine(v);
```

Cannot implicitly convert type `string' to `int'

What About 2nd App?

- Should it not be the same thing?
- We have a statement that initializes to an integer value
  - That much is known at compile time
  - Compiler knows that the hard-coded value of 36 is an integer, and it's what it's assigning to the variable
- Predictions:
  - Will we get the same thing (compiler error)?
  - Will it compile and run but ultimately crash?
  - Will it compile and run without crashing? If so, what's the output?
  - (get votes in class, then demo code to find out)

What About 2nd App?

- Output is “36Hello”
  - Somewhat surprising or no?
- Didn’t fail
  - But can we get similar code to fail?
- Is everything just going to magically work for dynamic typing?
- Let’s try to get it to fail:

```
dynamic d = 36;
d += (new MemoryStream());
Console.WriteLine(d);
```

What About 2nd App?

- Code fails with RuntimeBinderException:

```
dynamic d = 36;
d += (new MemoryStream());
Console.WriteLine(d);
```

- Makes sense that this would fail given the nature of what we're trying to do (add MemoryStream to int)
- Let's investigate why by first reviewing what dynamic typing is all about

Dynamic Typing in C#

- From MSDN:

"The **dynamic** type enables the operations in which it occurs to bypass compile-time type checking. Instead, these operations are resolved at run time.
The **dynamic** type simplifies access to COM APIs such as the Office Automation APIs, and also to dynamic APIs such as IronPython libraries, and to the HTML Document Object Model (DOM)."

More Broad Definition

- Dynamic typing just really means resolving access to an object's properties/methods at runtime instead of compile time.
- Some languages (like C#) are considered for the most part to be statically typed, but offer the dynamic keyword to simulate dynamic typing
- Other languages (like JavaScript) are dynamically typed
  - You really can't declare a type for a variable in the code.

What Does This Mean?

- For coding:
  - If we declare a variable with type dynamic, we can access any property or method and it will compile

```
dynamic variable = 101;
variable.DoFunctionThatDoesNotExist();
```

- That code WILL compile
- Obvious to us: Variable is an integer and will not have that member function
  - But compiler allows it
  - Crash at runtime when binding fails

What Does This Mean?

- For testing:
  - Serious implications here: code that is utter nonsense can compile and “will” crash at runtime
- “Will” in quotes for a reason:
  - The nonsense code from the previous slide may NEVER crash?
  - Why not?
  - (discuss, answer is simple)

What Does This Mean?

- For testing:
  - Serious implications here: code that is utter nonsense can compile and “will” crash at runtime
- “Will” in quotes for a reason:
  - The nonsense code from the previous slide may NEVER crash?
  - Why not?
  - It may never get run
- Usually code isn’t written into the product without the intention of running it at some point
  - But think about a bunch of tests trying to find dynamic-typing-oriented bugs

Testing Dynamically-Typed Code

- Tests that aim to test code with dynamic typing must hit the code path where the dynamic typing is used
  - True of all testing: if we want to find the bug, we'll need to hit the code path where the bug is
  - Low percentage paths make this more challenging if we're not careful with our tests

Testing Dynamically-Typed Code

- If code is of this form then our tests need to be engineered to hit all code paths

```
if (48% chance condition) { code without D.T. }
else if (48% chance condition) { more code without D.T. }
else { code with dynamic typing }
```

- Need to have tests that hit that 4% probably code path

Testing Dynamically-Typed Code

- Imagine the bug was:
  - someDynamicObj.Myfunction();
- when the correct version was:
  - someDynamicObj.MyFunction();
- One of the major points in all this:
  - A simple typo would have been caught by the compiler without dynamic typing
  - Not so with dynamic typing
  - Something we once got for free from the compiler now needs focused testing efforts
  - A reminder: test wouldn't fail if it didn't hit the code path where the typo resides

Testing Dynamically-Typed Code

- Options (not mutually exclusive):
  - Specifically write tests to hit all code paths
    - Use code coverage tools if available. These check to see what percentage of your code was hit after all your unit tests run. The goal is generally 100%.
  - Don't use dynamic typing
    - It really IS an option for some languages and some types of apps. As the MSDN article quote mentioned, it could make certain actions easier when writing the code, but it may complicate the testing effort. Like so many things, it's a tradeoff.
  - Use more advanced testing tools/methods
    - This is where testing starts to get "theoretical"
    - Enter: Static analysis

Static Analysis

- What is it?
  - Short answer: testing the code without actually executing it
  - Often even considered to be done without actually compiling it, although that's often only partially true
    - Reality is the code may be partially compiled
    - Yes, you can "partially compile" code and we're not talking about compiling half the code then stopping. More like halfway compiling all the code then stopping. Requires a discussion of compiler theory, which comes later.
- You may ask: "Does the compiler do this?"
  - Yes, in fact the act of compilation serves as a form of static analysis.
  - Compiler finds simple things like typos in property/method/field names. It didn't have to run the code to find a problem with it.

Static Analysis

- Why was it referred to as “theoretical” on an earlier slide?
  - Potential for this type of testing is far from realized
  - Not a lot of major IDEs are really providing static analysis tools that do powerful testing
- There are things that make the future look more hopeful
  - Open source compilation tools that let you use the compiler engines
  - These engines can provide powerful data structures for static analysis
  - In the next few years we may see some significant new tools and options for static analysis

Static Analysis

- Why is it being introduced with dynamic typing? Is it something only useful when using dynamic typing?
  - Absolutely not. It can be used for any type of code including 100% statically typed code.
- It just so happens that dynamic typing, being something that in a sense makes our compiler's error-checking logic less useful, is something that naturally leads to the idea of static analysis.
- Let's go back to dynamic typing, considering the typo bug on a low-probability code path and ask:
  - How do we do smart tests for these types of things?

Tests for Dynamic Typing Bugs

- We already discussed the option of intentionally writing unit tests to hit all code paths, using code coverage tools if we can.
  - Probably worth doing, since you gain some confidence if your test code coverage is high
  - But need a large testing effort
  - And again through all this we have to acknowledge that if it's a simple typo, achieving 100% code coverage in unit tests to find it is seemingly more work then we should have to do
- But consider if we could test the code itself, and not the compiled runtime version of the code

Tests for Dynamic Typing Bugs

- What if we could have test code that goes through product code (i.e. some type of smart code parser perhaps) and do something like this:

```
foreach (statement S in all_product_code) {
  if (S.IsObjMemberFuncCall) {
    CheckAllFunctionsInProductCodeForName(S.FuncName);
  }
}
```

Tests for Dynamic Typing Bugs

- The type of code on the previous slide could easily find typo bugs
- If we're looking through ALL function call statements in the code and comparing names against ALL functions declared in the code, we'll find if there's a match or not
  - Still potential issues if the object at runtime doesn't actually have that member function, but this is at least a step in the right direction

Tests for Dynamic Typing Bugs

- Two things to note with respect to typical statically-typed language compilation

1. What the compiler does ultimately does involve finding all functions in the code and finding all function calls as well. With dynamic typing it still has to generate a call (to a dynamic language runtime (DLR) function) when you do statements like: obj.CallFunc()
2. So if the compiler is doing all that work, why can't it do what's on the previous slide? What's the compiler actually doing when you call a member function on a dynamic object?

Dynamic Object Function Call

- A statement of the form:

dynObj.CallMyFunc(64);

- Gets translated by the C# compiler to something like:

```
var func = DLRLib.GetPtrToFuncNamed(“CallMyFunc”);
if (null == func) { throw binding exception }
else func.Invoke((int)64);
```

Dynamic Object Function Call

- Compiler and dynamic typing system is not designed to check through ALL code at compile time to see if that function might actually not exist
- Only required to make a call to the DLR stuff to lookup and execute that function
- Works similarly in many (most/all?) dynamic typing scenarios
- The contract is supposed be:
  - Programmer writes code that only accesses members known to exist in the dynamic type
  - Programmer needs to write code to ensure that the function/property/field exists
  - DLR merely looks it up at runtime based on the name the programmer typed into the code

Back To Static Analysis

- So the question now is: what types of static analysis can we perform?
  - Do we have to write code parsers?
  - Can we access what the compiler knows as it's compiling and somehow test off that?

Back To Static Analysis

- So the question now is: what types of static analysis can we perform?
  - Do we have to write code parsers?
  - Maybe. If one doesn't have access to some assistance from the compiler, then a static analysis tool may actually parse the actual language code.
  - Can we access what the compiler knows as it's compiling and somehow test off that?
  - Also maybe. If we can use a compilation engine then we can get access to data structures representing the code. These can be used for static analysis. This also sends us into a compiler theory discussion. (on whiteboard in class)

If We Need to Parse Code…

- Antlr
  - http://www.antlr.org/
  - Parser generator
  - From a grammar you can generate code that parses other code
  - Just one of many options out there
- We're not as concerned with what compiler generator you might use, but what you could get from it and how you could use it for static analysis
  - (investigate on whiteboard in class)