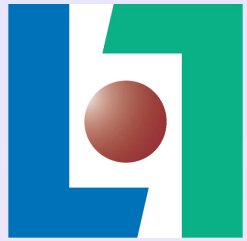

Choosing a programming language

Robert Strandh

University of Bordeaux

May, 2018

Dfind, Göteborg

Overview of talk

- Programming language characteristics.
- Common misconceptions.
- Requirements for making a good choice.
- Risk analysis.

How the choice is often made

The choice of programming language for a project (when there are several possibilities) is often based on *gut feeling*.

Often, no real analysis is made, because the decision maker:

- has only partial knowledge of the characteristics of possible choices;
- sometimes has incorrect information about the possible choices;
- has insufficient training to appreciate the characteristics of possible choices;

How the choice is often made

Often, no real analysis is made, because the decision maker:

- has insufficient experience with the possible choices to determine which choice is adapted to the current project;
- has insufficient information about the cost associated with training staff in a new language vs the productivity advantages of that language;
- has insufficient information about the cost associated with hiring new staff for a new language vs the productivity advantages of that language.

Language vs implementation

Language vs implementation

Language: A description of the syntax and semantics of conforming programs, and of consequences of using non-conforming constructs.

Example of the latter: In C, obtaining a pointer outside of an array has undefined consequences. (It is interesting to contemplate why.)

Language vs implementation

Implementation: Software that accepts conforming programs and executes them according to the language semantics, and that reports non-conforming constructs where required by the language definition.

Language vs implementation

In many cases, the language definition does not require the compiler to check for non-conforming constructs. Why?

As a consequence, many non-conforming programs go undetected.

Language vs implementation

The distinction language/implementation is sometimes blurred:

- Some languages are defined by a *reference implementation*. Examples?
- Some language definitions are controlled by the same organization that supplies some dominating implementation. Examples?

Language vs implementation

The distinction language/implementation is sometimes blurred:

- Some languages are defined by a *reference implementation*. Perl, Python, Scala, Ruby, Clojure.
- Some language definitions are controlled by the same organization that supplies some dominating implementation. Java, C#, Objective C.

Language characteristics

Strong vs weak typing

A language can be *strongly typed* or *weakly typed*, and even *untyped*.

Strongly typed: It is impossible for an object of one type to be mistaken for an object of a different type. Either the compiler made sure no mistake is possible, or the run-time system checked it. Examples?

Weakly typed: No such guarantees are made. Examples?

Untyped: Data can be interpreted differently by different operations. Examples?

Strong vs weak typing

A language can be *strongly typed* or *weakly typed*, and even *untyped*.

Strongly typed: It is impossible for an object of one type to be mistaken for an object of a different type. Either the compiler made sure no mistake is possible, or the run-time system checked it. Examples: Java, C#, Common Lisp, Scheme, Clojure.

Weakly typed: No such guarantees are made. Examples: C, C++.

Untyped: Data can be interpreted differently by different operations. Example: Assembler.

Static vs dynamic typing

A language can be *statically typed* or *dynamically typed*.

Static typing: Type information is associated with the *variables* in the program. Type checking or type inference is always handled at compile time. Examples?

Dynamic typing: Type information is associated with the *objects* manipulated by the program. Type checking must sometimes be done at execution time. Not always? Examples?

Static vs dynamic typing

A language can be *statically typed* or *dynamically typed*.

Static typing: Type information is associated with the *variables* in the program. Type checking or type inference is always handled at compile time. Examples: C, Java, ML, Haskell.

Dynamic typing: Type information is associated with the *objects* manipulated by the program. Type checking must sometimes be done at execution time. Examples: Lisp, Scheme, Python.

Manifest vs implicit typing

A statically typed programming language may be based on *manifest typing* or *implicit typing*.

Manifest typing: The programmers supplies the type of the variables explicitly. Examples?

Implicit typing (sometimes called *latent* typing): The compiler *infers* the type of the variables from the operations it participates in. Examples?

Manifest vs implicit typing

A statically typed programming language may be based on *manifest typing* or *implicit typing*.

Manifest typing: The programmers supplies the type of the variables explicitly. Examples: C, C++, Java, C#.

Implicit typing (sometimes called *latent* typing): The compiler *infers* the type of the variables from the operations it participates in. Examples: ML, Haskell.

Manifest vs implicit typing

Why is the distinction between manifest and implicit typing important?

With manifest typing, the programmer is given too much responsibility too early in the development process. Choices made may easily change later on.

Manifest vs implicit typing

Some dynamically typed programming languages allow optional type declarations.

Such declarations are sometimes used in order to allow the compiler to generate faster code.

Static vs dynamic

A programming language can be either *static* or *dynamic*.

Static: There is a clear distinction between *compile time* and *run time*. Code is *generated* at compile time and *executed* at run time. Examples?

Dynamic: There is no clear distinction between *compile time* and *run time*. The program might change as a result of executing code (for example, executing a statement that defines a function or a class). Examples?

Static vs dynamic

A programming language can be either *static* or *dynamic*.

Static: There is a clear distinction between *compile time* and *run time*. Code is *generated* at compile time and *executed* at run time. Examples: C, C++, Java, C#, Haskell, ML.

Dynamic: There is no clear distinction between *compile time* and *run time*. The program might change as a result of executing code (for example, executing a statement that defines a function or a class). Examples: Common Lisp, Scheme, Python.

Interpreted vs compiled

Can you guess what an *interpreted programming language* is and what a *compiled programming language* might be?

Interpreted vs compiled

Can you guess what an *interpreted programming language* is and what a *compiled programming language* might be?

A programming *language* is neither interpreted nor compiled. A programming language *implementation* is either one or both (it is a spectrum of possibilities).

Interpreted vs compiled

The distinction is important because it affects the *execution performance* of the program.

An implementation that compiles to native code has the potential of generating *fast* code.

An implementation that has more elements of interpretation can generate code that is slower by a factor 10 or more compared to code generated by a native compiler.

Manual vs automatic memory management

Manual memory management: The language requires the programmer to de-allocate objects that are no longer going to be referenced. Examples?

Automatic memory management: The implementation of the language is required to automatically recycle objects that are no longer referenced. Examples?

Manual vs automatic memory management

Manual memory management: The language requires the programmer to de-allocate objects that are no longer going to be referenced. Examples: C, C++

Automatic memory management: The implementation of the language is required to automatically recycle objects that are no longer referenced. Examples: Java, C#, Common Lisp, Haskell.

Standardization

A language is said to have an *independent standard* if and only if the definition of the language is published by an organization other than a supplier of an implementation.

Scripting languages

There are few properties that characterize scripting languages. Probably only:

- The creators meant for the language to be used for scripting.
- It is a dynamic language.

Common scripting languages are single-implementation languages with the implementation written as a slow interpreter. This technique is considered acceptable because of the first item above.

Scripting languages

Typically, a static language is used for the main body of code, and a "scripting language" for, um, scripting.

When the advanced user starts writing serious code using the scripting language (because that's the only choice possible), the combined result is slow despite the best intentions of the creators.

Furthermore, debugging code written in two languages is typically hard.

Domain-specific languages

A domain-specific language is a language that was designed for a particular family of programming tasks.

Often, the language is defined by the same organization that then uses it.

The productivity advantage can be huge.

Designing and implementing a domain-specific language requires expertise in language design and compiler technology.

Common misconceptions

Manual vs automatic memory management

Common misconception: manual memory management is faster than automatic memory management.

With manual memory management, for modularity, it is often necessary to copy objects or to use reference counters.

Such techniques can easily incur a performance penalty of a factor 10–100 on modern hardware. Why?

Modern garbage collectors are fast, concurrent, multi-threaded, and some have real-time or near-real-time guarantees.

Compiled/static vs interpreted/dynamic

Common misconception: dynamic languages must be interpreted.

Conversely: only static languages can be compiled.

Modern implementations of dynamic languages generate native code *on the fly*.

Alternatives to scripting languages

Use the same dynamic language for the main body of code and for scripting purposes.

Choose a language that has an efficient implementation that compiles to native code.

So how do we choose a language?

So how do we choose a language?

Making a good language choice requires:

- Good knowledge of the characteristics of several programming languages (the subject of this talk).
- Good knowledge of the nature of the task to be accomplished.
- A separate and detailed budget for each "reasonable" language choice.

We will look at the last item a bit more.

How not to choose

- "We need all the speed we can get, and it is known that the C++ compiler generates very fast code. Therefore we choose C++."
- "All our programmers already know Java. Therefore we choose Java."
- "We have made a huge investment in programming tools for C#. Therefore we choose C#."

What to include in the budgets

- Cost of acquisition of language tools.
- Estimated development and maintenance cost.
- Cost of training and hiring new staff.
- ...
- Risk analysis.

Risk analysis

For every major possible choice (tools, staff, development method, etc.), make a list of possible events that might have a negative impact on the project.

For each event, state:

- its likelihood,
- the cost to the project if nothing is done,
- actions to avoid the negative impact, and
- the cost of those actions.

Risk analysis

Example:

Choice: Make Joe a member of the staff. Joe is a reckless driver.

Event: Joe has a traffic accident and can no longer work on the project.

Likelihood: Unlikely

Cost if nothing is done: The project will be delayed by six months.

Action: Hire a replacement for Joe.

Cost of action: Salary, training, etc.

Risk analysis

Example:

Choice: Use the language C#.

Event: Microsoft is bought by Apple (or Google) and C# is no longer supported.

Likelihood: Unlikely

Cost if nothing is done: All code must be rewritten in Java.

Action: Obtain (buy, develop) a replacement for Microsoft C#.

Cost of action: Cost of purchase or development.

Standardization and risk analysis

If a language does not have an independent standard, its specification may change as a result of the organization that supplies the implementation.

The cost to a project could be huge. Much code may need to be rewritten, perhaps not immediately, but over time.

Such possibilities must be taken into account in the risk analysis.

Examples?

Standardization and risk analysis

If a language does not have an independent standard, its specification may change as a result of the organization that supplies the implementation.

The cost to a project could be huge. Much code may need to be rewritten, perhaps not immediately, but over time.

Such possibilities must be taken into account in the risk analysis.

Examples: Java, C#, Python, Scala, Ruby.

Availability of good implementations

A good implementation may exist when a project is started, but might then be abandoned over time.

Again, the cost to a project could be huge, including a complete rewrite using a different language.

Such possibilities must be taken into account in the risk analysis.

Examples?

Availability of good implementations

A good implementation may exist when a project is started, but might then be abandoned over time.

Again, the cost to a project could be huge, including a complete rewrite using a different language.

Such possibilities must be taken into account in the risk analysis.

Examples: Objective-C, Dylan.

Availability of programmers

New graduates may only know a few languages.

They may also have the wrong idea of languages they do not (yet) know and sometimes also about the languages they (think they) know.

For other languages, it may be necessary to train existing programmers or to hire new programmers.

The salary may need to be higher. That will be another factor in the cost analysis of the project.

Availability of programmers

For a medium-sized or large company, it is advisable to have programmers with knowledge of several different programming languages using different programming paradigms.

That way, a large spectrum of languages can be covered.

Programmers can participate in decisions about programming languages, and they can help train other programmers.

Conclusions

Choosing the right language for a task can have a great impact on the amount of work it takes to finish that task.

Making the right choice requires expertise that must either be developed in-house, or hired when a new project is starting.

It is best to do a detailed cost/benefit analysis, including a risk analysis, for each reasonable choice of a language.

Questions?

Thank you for listening!

Do you have any questions?