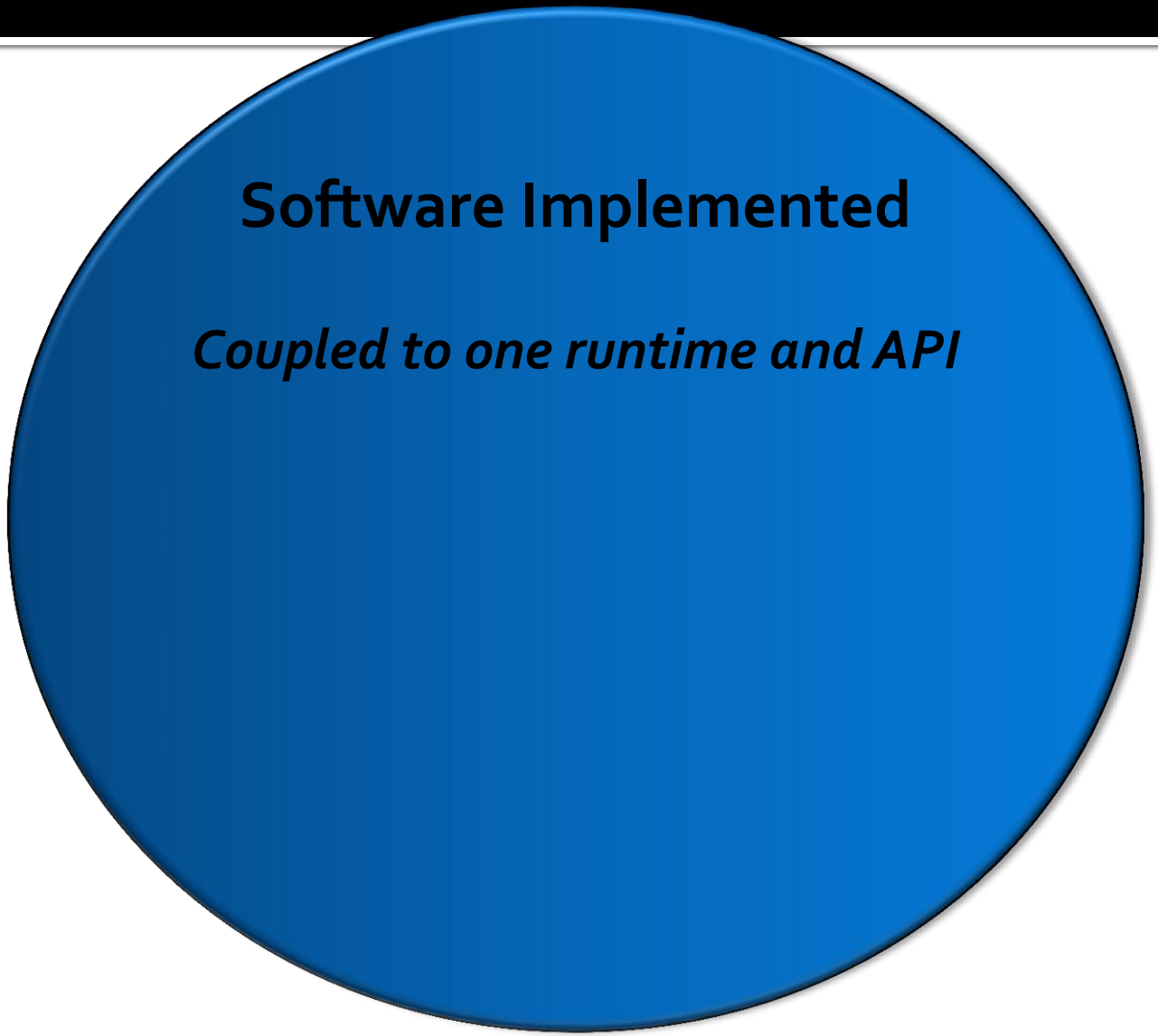
Alexis Ferreyra

Semantic Programming with LayerD

PyCon Argentina 2012

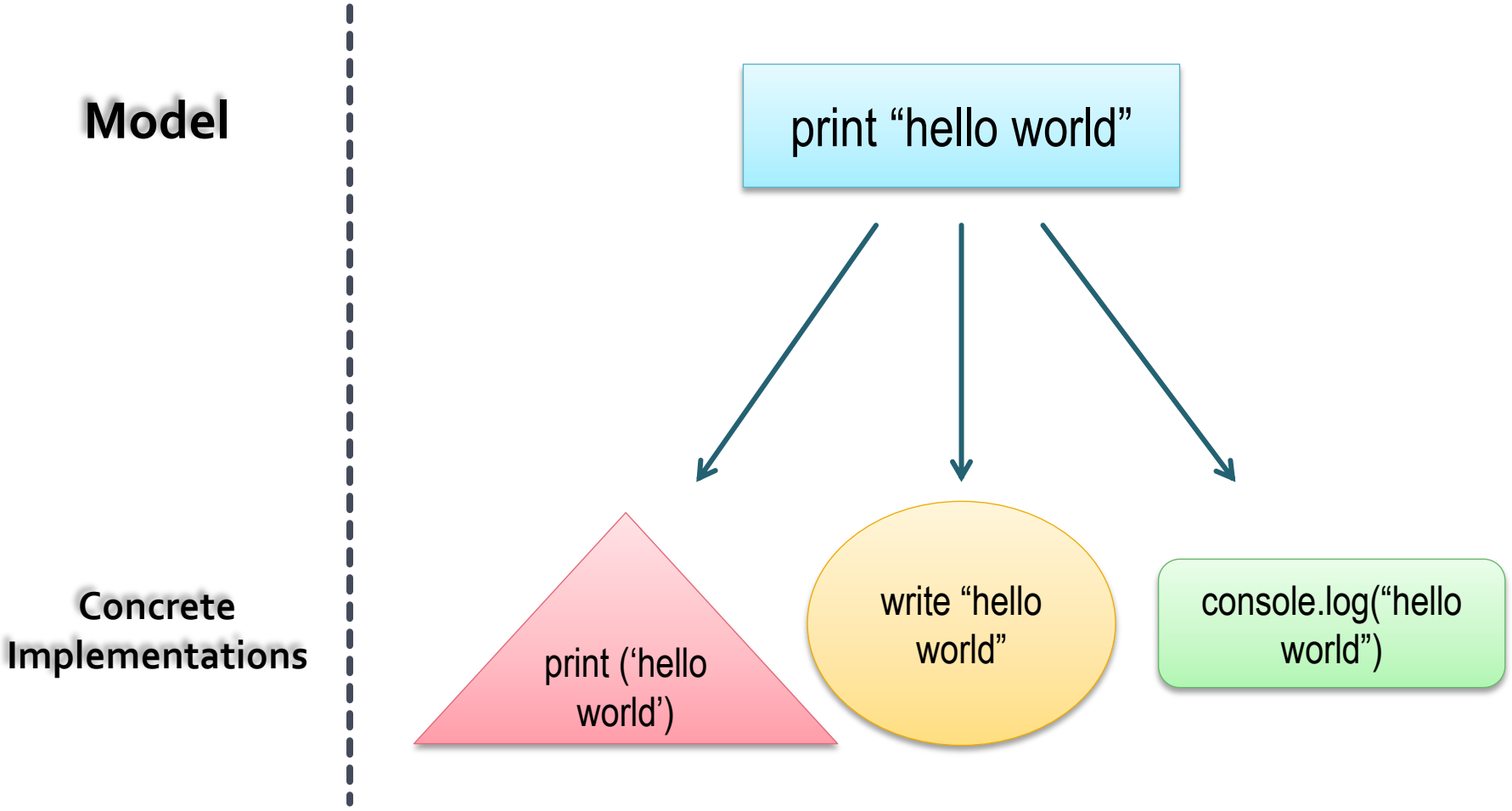
Software as an Abstraction



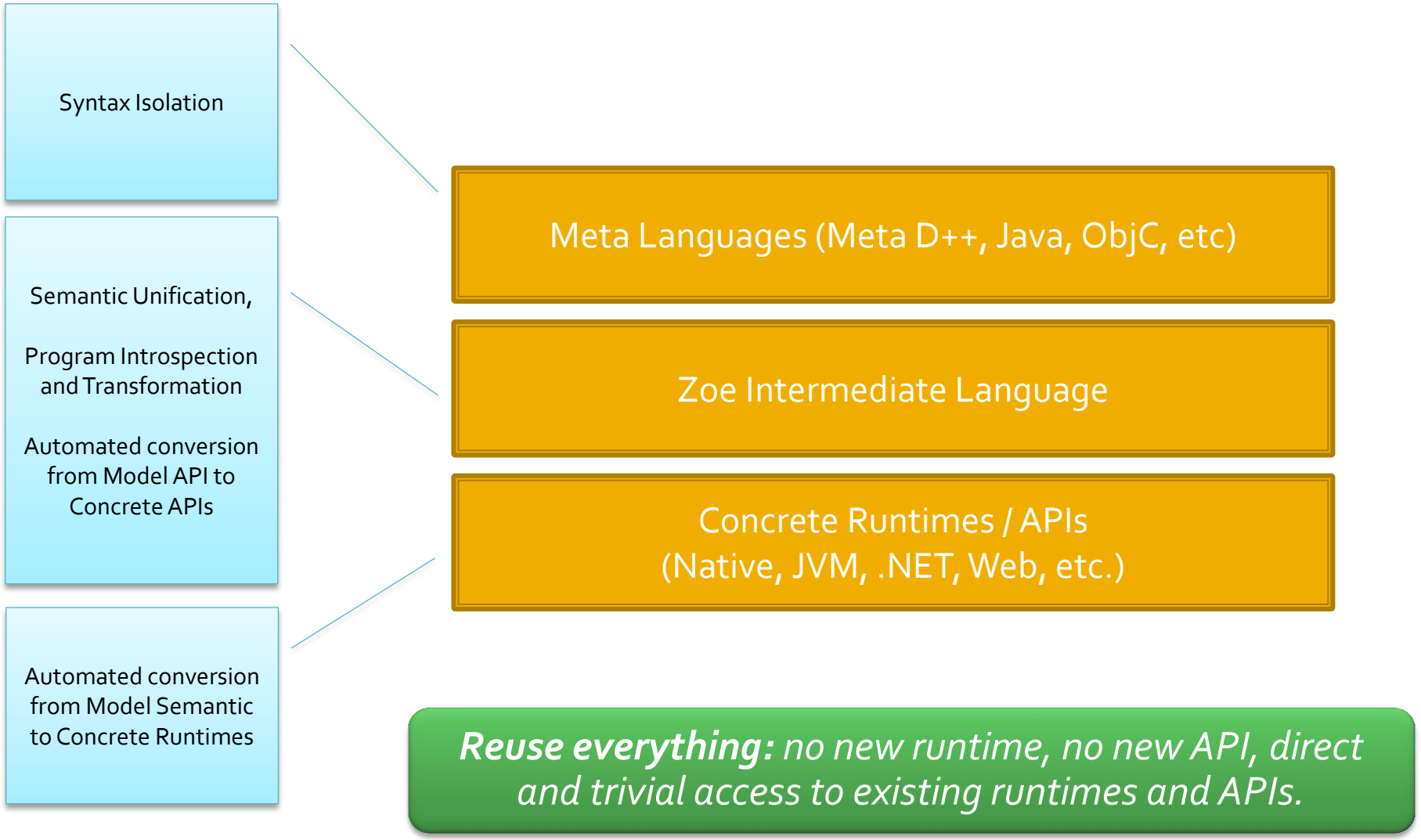
Why?

- To make software reusability an order of magnitude better
  - Reuse source code and knowledge about API by decades or centuries
- To free developers from learning again, and again, and again to write “hello world”
- Because the world has changed since 1960

Semantic Programming?

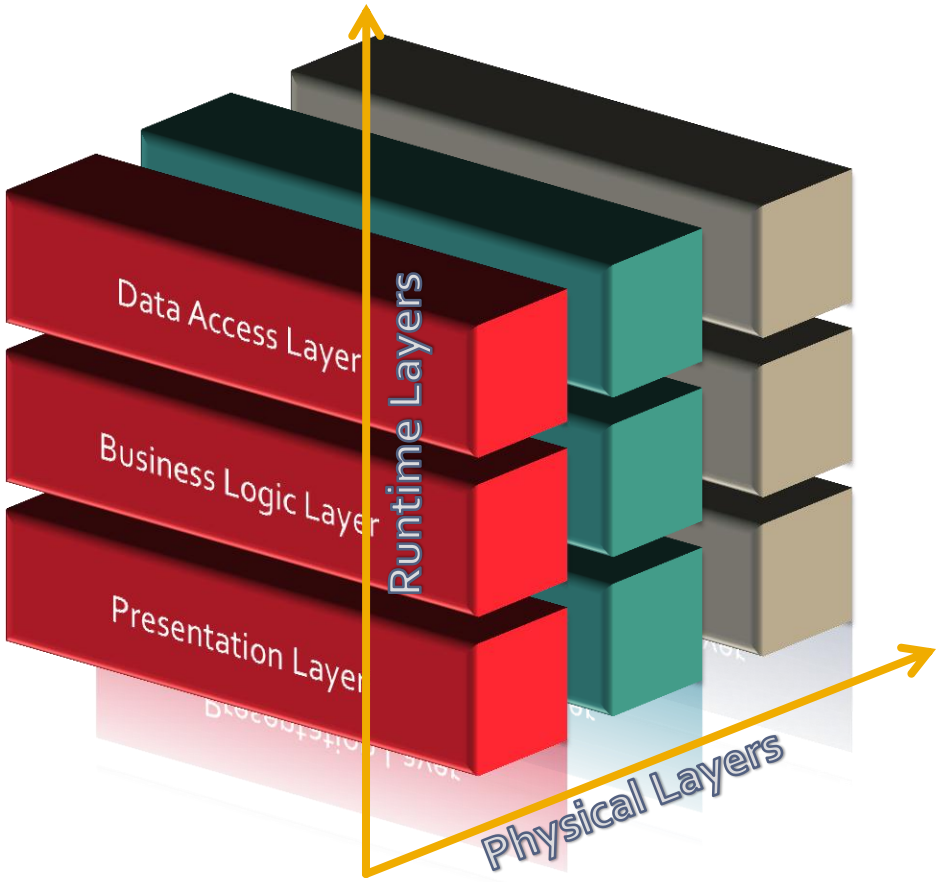


LayerD Architecture



Put LayerD at good use

Abstract your source code using bi-dimensional layers



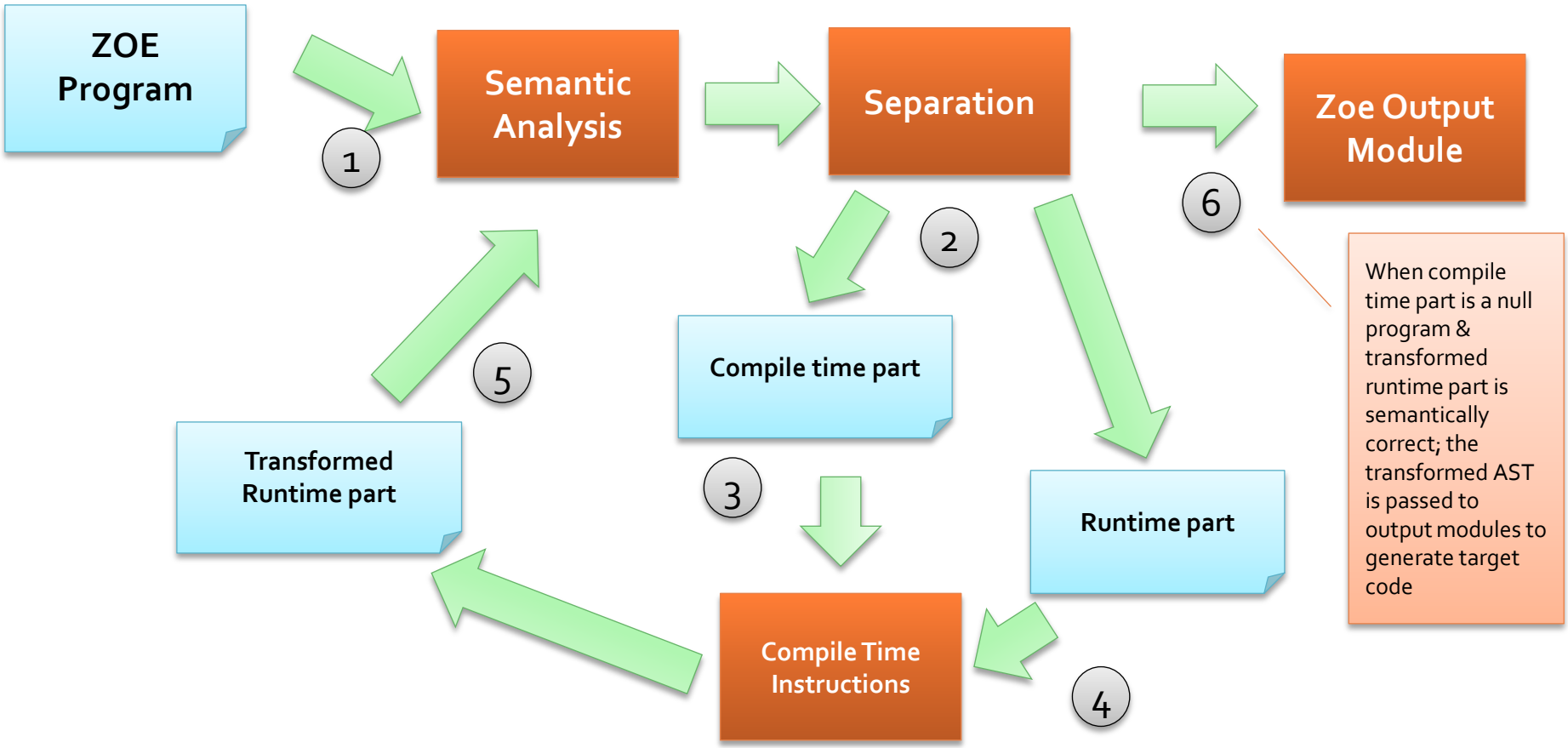
Physical Layers:

- Abastract APIs at compile time
- Abstract Runtime semantic with Zoe code generators

LayerD core: The Zoe Intermediate Language

- Object Oriented general purpose language
- Hides underlying semantic by providing an unified semantic view
- Multistage compiler with programmable compile time
- Full compile time introspection and program transformation
- Compile time API encapsulation
- Template and DOM based code generation
- Gracefully degradation of target runtime features
- Modular code generation

Zoe Compile Time Cycles



Demo Time

LayerD's Current State

- Open source project at
  - http://code.google.com/p/layerd/
- Code Generators for C#, Java, C++, Javascript, Python
- Used on commercial products
  - A project for Microsoft in 2009
  - A project for Gobierno de Córdoba 2010
  - A new tool for Intel (coming soon early 2013)
- Contributors:
  - Intel
  - Myself ☺
- TODO
  - Better documentation
  - Builds for Windows and Linux
  - Develop a community

It will take decades until these concepts are taken by the industry ☹

You can help by raising awareness

Thanks

http://code.google.com/p/layerd/wiki/Home

alexis.ferreyra@gmail.com

BACKUP

What's LayerD?

- A small framework to develop abstract software
  - Software that lives for centuries, literally speaking
- **Goal:** Instead of providing another cross-SDK give to programmers the right tools to make their source code portable
- LayerD's timeline
  - 2002/2003: Project conception and high-level design
    - Project hibernation until late 2005 ☹
  - 2005-2007: First Implementation at UTN-FRC
  - 2008-2009: First real-world applications
    - A graphical DSL project developed for Microsoft at UTN (http://archive.msdn.microsoft.com/emall/Release/ProjectReleases.aspx?ReleaseId=2637)
    - DSL for DAL & database in a system for Government of Cordoba
  - 2010: Core component of an Intel Tool (public release coming soon)

Examples of Zoe Infrastructure Usages (*see backup for further details)

- One high level language to program for multiple runtimes
- Compile Time API Encapsulation by transformation (used in App Porter for API transformation)
- Implementation of text based DSL

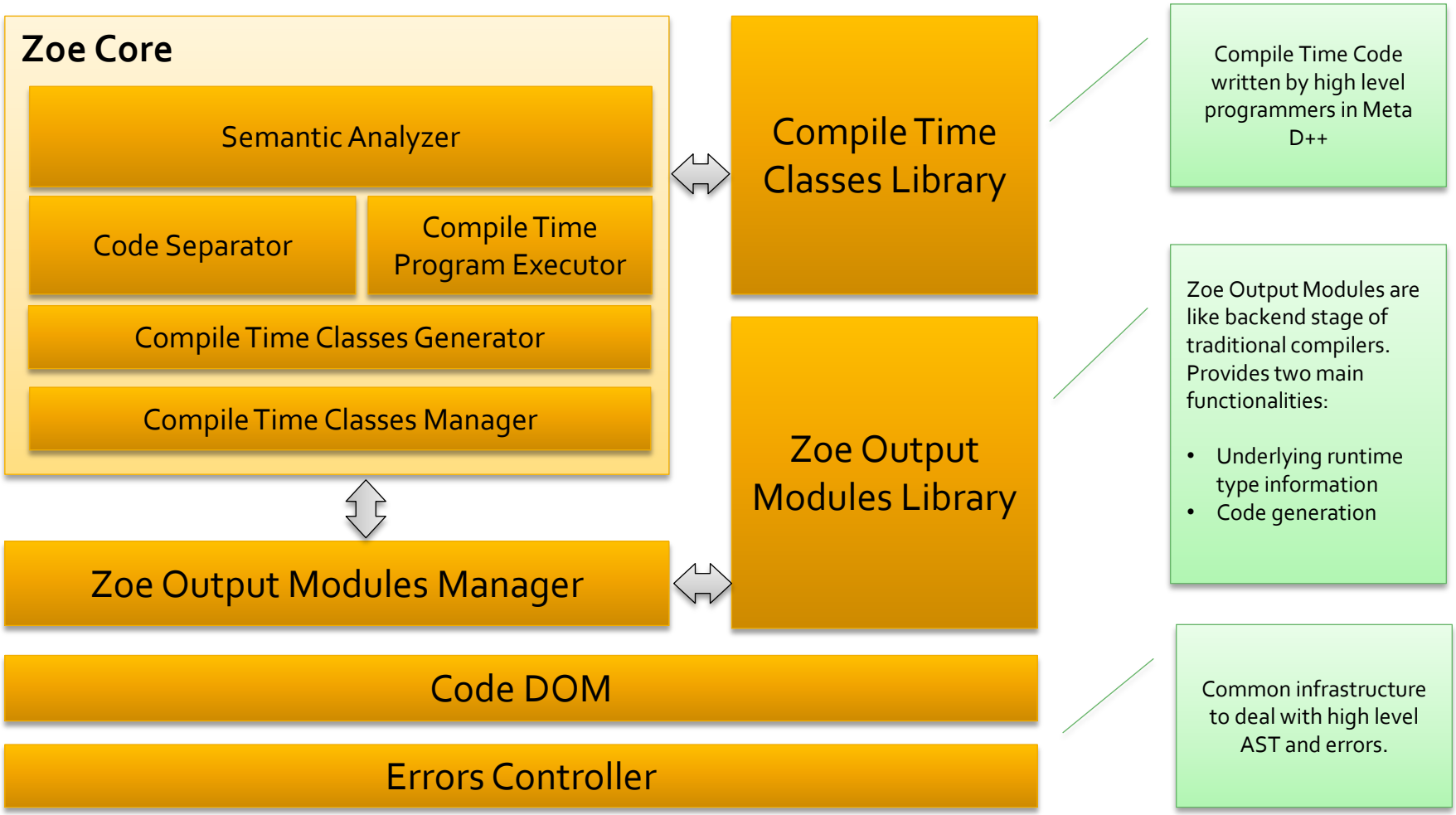
- User defined semantic structures
- Program introspection and analysis
- Code Generation for multiple platforms reusing templates and logic
- Interactive Compilation for automatic source code update, refactoring, etc.

Key learning's from LayerD

- Abstraction of Source Code itself is key and powerful
- Having an unified view of underlying runtimes semantics helps you think about portability in early stages
- Embedded DSL are great for growing software and architectures
- Zoe compile time API encapsulation is good enough for source-to-source transformation
- Ramp-up new "languages" (Meta D++ syntax) is hard*. Much better to use existing ones as syntax facades.

*may be it's because it's ugly or because it's just myself pushing ☺

Zoe Compiler Architecture



Programming Languages

Facts:

- Software is an abstraction while in the design board
- Source code is a live and always evolving creature
  - Runtimes used to run a software changes through time
  - API used to implement a software evolves and changes through time
  - Programming languages features changes
- Modern software runs in more than one runtime
- Syntax is a matter of style, semantic is what define what a language can and can't do
- Source code base is one of the most valuable assets of any software company

Still, most programming languages:

- Don't provide the tools to abstract runtimes
- Are designed coupled with one runtime
- Don't provide means to update API usage
- Couple syntax with semantic
- Doesn't allow to be extended in a controlled fashion
- Not always provides means to abstract source code itself

Typical Implementation Problems in Software Engineering

- You need some runtime + API in order to implement. But you also need a means to abstract them.
- Software dependencies are always evolving entities:
  - API changes, becomes deprecated, etc.
  - One runtime is not always suitable for your need, becomes obsolete with time, etc
- If you choose one runtime & API you are artificially crippling the reach of your implementation
- Programming languages are designed coupled with one runtime in mind. Still, modern software runs in more than one runtime
- Life of a source code base is artificially limited by the life of used APIs and runtime
- As a programmer, you have to learn multiple times how to do the same thing
- Cross-platform SDK, most of the time, are cross-platform APIs. Can't be extended to support new runtimes.

LayerD goal is to provide the required tools & languages to deal with all of these problems

Meta-languages

- Syntax facades to intermediate Zoe language
- Are compiled to Zoe
- Current meta-languages:
  - Meta D++ : allows access to all features of Zoe (http://code.google.com/p/layerd/wiki/Hello_World_Samples)
  - Limited support:
    - Java 1.4 (from PoC for App Porter)
    - C/ Obj-C (from App Porter)
    - Argentino (used to be a Spanish based language, now it's abandoned ☹)
- Planned
  - Sub-set of C#

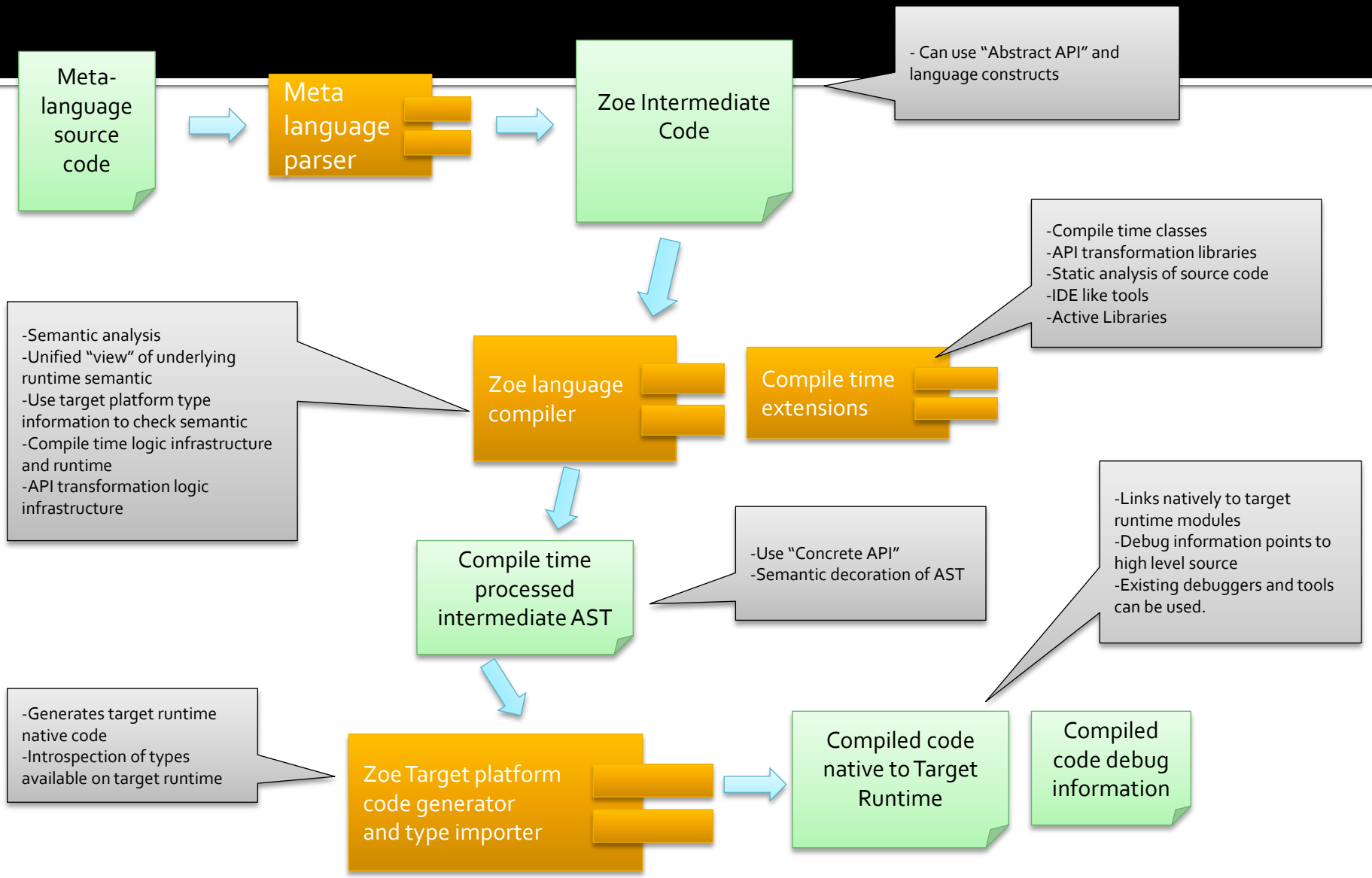
Zoe Compile Time Classes

- They are ordinary Classes with access to following infrastructure:
  - Introspection of program being compiled
  - Transformation of program being compiled by using a DOM and template programming
  - Straightforward transformation of API usages
  - Templates:
    - are written using high level Meta-languages
    - don't need to know details about underlying runtime
    - Template variables are typed
    - Injected code is typed and checked for static semantic correctness in the next compile time
  - Type information of underlying runtimes is available in Zoe unified way
  - Interface with Zoe Core module allows to
    - Emit and detect compile time errors
    - Introspect types table information
    - Query about target runtime capabilities
  - Compile Time Classes are compiled and added as plug-ins into the compiler.
    - Compile Time Classes themselves can use other compile time code when compiled ☺

LayerD development status

- Meta D++ parser ~90%
  - Still, Meta D++ language needs a design cleaning with lessons learned
  - Parser implemented using C++/BtYacc is slow for large files
- Zoe Compiler ~75%
  - Some semantic checking's are not implemented
  - Compiler Code is linear, not parallelized
  - Missing last stage before moving into code generators (lower level AST generation)
  - Desired, but not yet implemented language features:
    - Hybrid classes (compile and runtime code in one class)
    - Native Closures (today you have to write a CTC to implement it)
    - Use dynamic code for CTC instead of statically compiled
    - Better compile time debugger
    - Better support for parameterized types
- Zoe Code Generators
  - C# code generator: stable but doesn't support all .NET new features
  - Java code generator: working but drafty
  - C++: working but not stable (App Porter Beta 1 QA), Missing type importer
  - JavaScript: under development for App Porter, Missing type importer

LayerD Compilation process

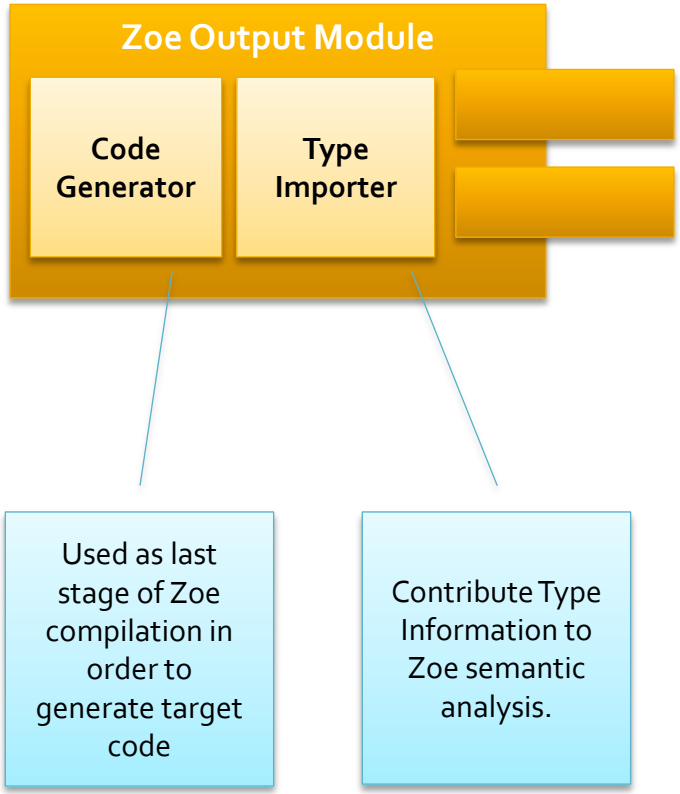


Zoe Output Modules

These modules provides meta-information about characteristics of target runtime, generates target code, and contribute type information of imported types.

Each Zoe Output Module contains two components:

- **Code Generator:** same than back-end stages of traditional compilers must generate executable/target code from intermediate code that is known to be semantically correct.
- **Type Importer:** provides types introspection for supported platforms. Process "import directives" and provides underlying runtime type information to Zoe Core



LayerD design guidelines

- Decouple syntax from semantic
- Don’t introduce any new API or Runtime
- Provide direct access to existing APIs
- Integrate with existing and future runtimes
- Integrate with existing tools and languages
- Provide means to extend the functionality of high level languages without needing to write new compilers and tools.

Compile time API encapsulation

```
this->myLabel = new QLabel();
this->myLabel->setGeometry(QRect(5, 10, 60, 60));
this->myLabel->setText(QString(""));
this->myLabel->setParent(this->getView());
```

Portion of code using **Qt** standard API

- *Object oriented transformations*
- *Automatic capture of function calls and types declarations from types signature*
- *Type safe compile time programmability with full reflection*
- *Complex transformations like change an interface and his implementation, change one statement by multiple ones, add new classes, methods, etc.*

DSL with Meta D++

- Use lexical structure of meta language
- Can be embedded with ordinary code
- Implemented in compile time classes
- Infrastructure for meta templates and DOM code injection
- Generated code is semantically checked

Sample DSL for UI

```
GUI::Window(SampleWindow){
  Text = "Simple Window with a DSL";
  Controls{
    Button(Exit){
      Text = "Exit"; AutoSize = true;
      Click{
        this.Close();
      };
    };
  };
};
```

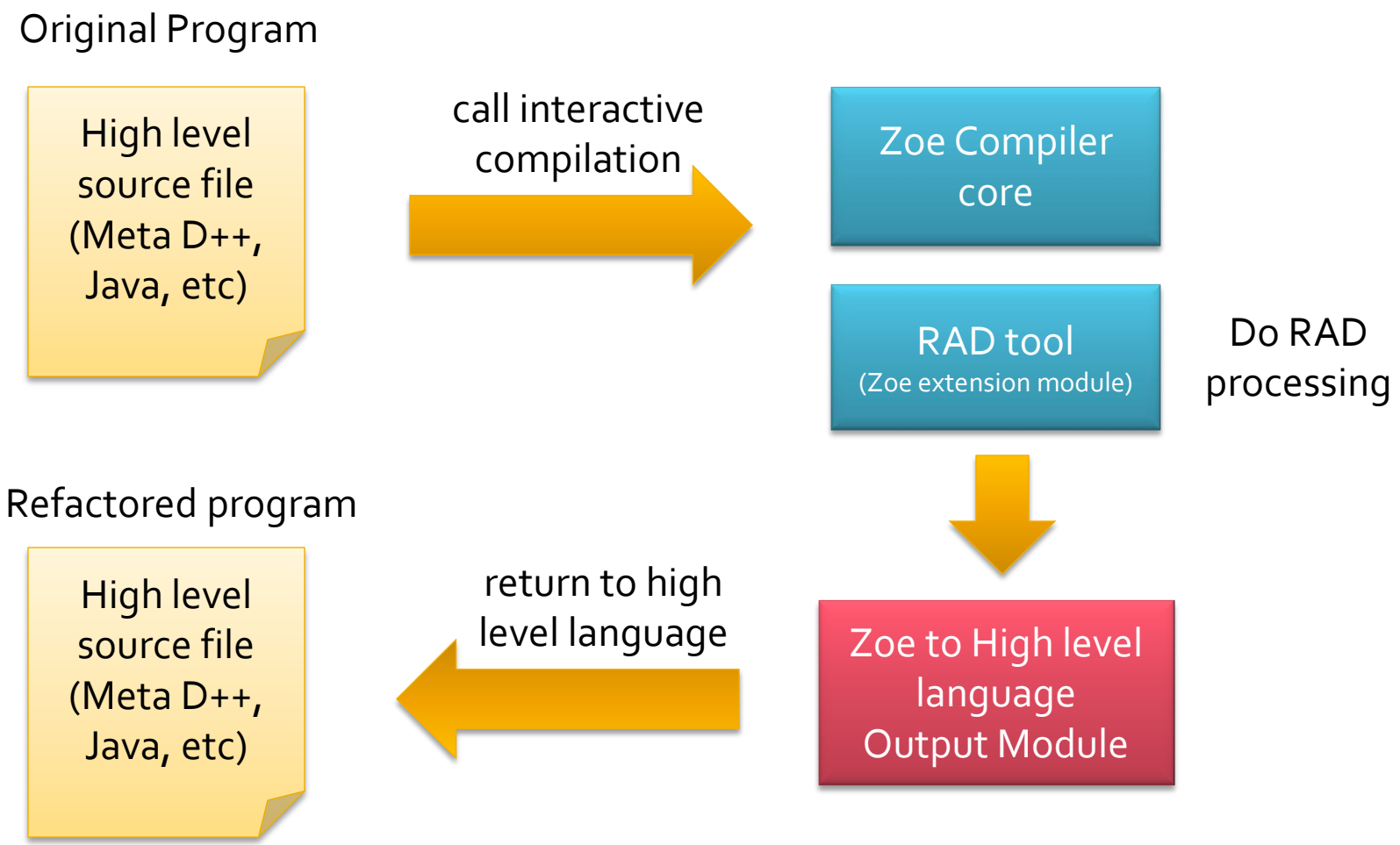
Actual DSL for API mapping used in App Porter

```
API::Define( SampleAPIString )
{
  Inherits = SampleBaseAPIObject;
  TargetType = js::SampleTargetTypeString;

  Property(samplePropertyName is string^)
  {
    Get
    {
      return writecode( ileft.getSomething() );
    };
  };

  Method(sampleTestMethod is int)
  {
    Parameters:
    {
      SampleAPIString* param1;
      SampleAPIString* param2;
      object va_list;
    }
    Code:
    {
      // ...
    }
  };
}
```

Interactive compilation of LayerD programs



Is it worth the effort?

Think about how many times you had to learn how to open and write a file, or to use a dictionary, or to write the same algorithm for that new runtime