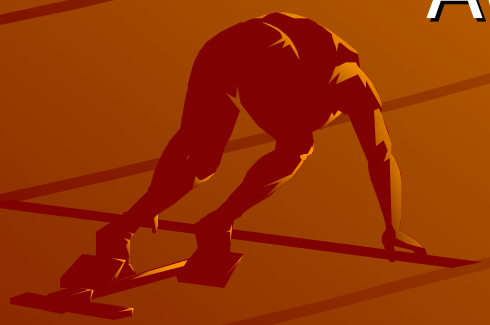
Bottom-Up Modeling

Agile Code Generators



Context

Name: Dave Whipp

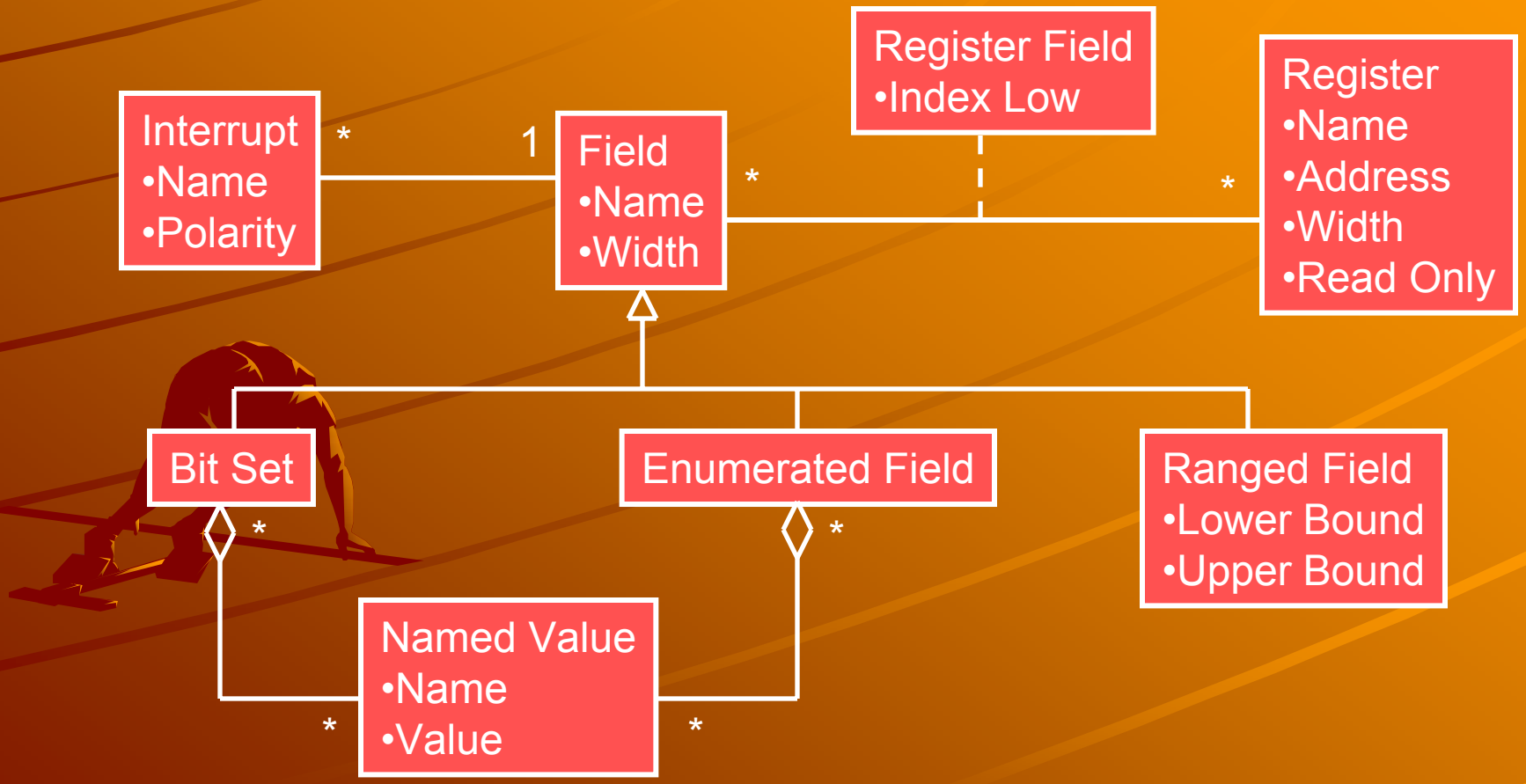
Company: Fast-Chip Inc.

Product: 10Gbps Network Services Processor

Role: Front-End Design Verification

People: Team of Six

Registers: Domain Model



The Software

- "C-model" of ASIC
  - Golden Model for design verification
  - Also to be supplied to customers
  - Run in embedded environment
    - Code in C for portability
- Cycle and Priority accurate for DV
- Functionally accurate for customers

“C6” Methodology

- Six people; Cost of bugs negligible
  - Avoid a-priori architecture decisions
  - The source code is the design
  - DTSTTCPW and YAGNI

- Agile Manifesto

See Also: “Agile Software Development”, Alistair Cockburn

Registers: “C” Representation

```
Uint64 register_read(int chip_id, Uint32 address)
{
    switch (address) {
                    …
        case ERX_TestModeReg:
                return test_mode[chip_id]->mode2_en << 1
                        | test_mode[chip_id]->mode1_en << 0;

        case ERX_MCTRP:
                    …
    }
}
```

Taxonomy of Code Generation

- Assembler (syntax mapping)
- Compiler (semantic mapping)
- Partial Assembler (many UML tools)
- Partial Compiler (Lex/Yacc)

- Pre-Processor
- Meta-compiler

- Wizard
- Round-Trip

Registers: “C” Representation

```
Uint64 register_read(int chip_id, Uint32 address)
{
    switch (address) {
                    …
        case ERX_TestModeReg:
                return test_mode[chip_id]->mode2_en << 1
                        | test_mode[chip_id]->mode1_en << 0;

        case ERX_MCTRP:
                    …
    }
}
```

Tools for Bottom-Up Modeling

- Refactorings:
  - Introduce Code Generator
  - Introduce Meta Data
- Analyze the Code
  - Variation Analysis
  - The Magnifying Glass
- Unification

Refactoring

- Change structure of existing code
  - Without changing its **external** behavior
- Well defined steps (mechanical?)
- Supported by tests (or proofs)

- Refactoring Code Generators
  - What is "External Behavior"?

Introduce Code Generator

1. Create Tests
2. Modify Build System
3. Create the Generator

4. Refactor the Generator

Introduce Code Generator

1. Create Tests
   - Copy .c (.h) to .c.gold

- Test will be "diff" against this file
  - Perfect Test!
  - Complete and Unambiguous
  - (But be prepared to change the test)

Introduce Code Generator

2. Modify Build System
   - Run .pl file to create .c.new
   - Diff .c.new .c.gold
   - Rename .c.new to .c

- Issues
  - Automatic build
  - Ability to be flexible

Simple Makefile

```
foo.c: foo.c.pl
    perl –w foo.c.pl > foo.c.new
    diff foo.c.new foo.c.gold
    mv foo.c.new foo.c
```

Introduce Code Generator

3. Create the Generator
   - Given foo.c
   - Create foo.c.pl

```
int main ( void )
{
    printf("hello, world\n");
}
```

```
print <<'HERE';
int main ( void )
{
    printf("hello, world\n");
}
HERE
```

Introduce Code Generator

4. Refactor the Generator
   - Duplication within a generator
   - Duplication between generators

   - Introduce Structure
   - Create Unit Tests
   - Work Middle-Out
   - Don’t change the output!

Registers: “C” Representation

```
Uint64 register_read(int chip_id, Uint32 address)
{
    switch (address) {
                    …
        case ERX_TestModeReg:
                return test_mode[chip_id]->mode2_en << 1
                        | test_mode[chip_id]->mode1_en << 0;

        case ERX_MCTRP:
                    …
    }
}
```

Registers: “C” Representation

```
Uint64 register_read(int chip_id, Uint32 address)
{
    switch (address) {
                    …
        case ERX_TestModeReg:
                return test_mode[chip_id]->mode2_en << 1
                        | test_mode[chip_id]->mode1_en << 0;

        case ERX_MCTRP:
                    …
    }
}
```

Registers: Code Generator

```
    …
    switch (address)
    {
        case ERX_TestModeReg:
\! loop ($name, $offset) ([“mode2_en”, 1], [“mode1_en”, 0])
\! loop-first { $prefix = “return”; $suffix=“” }
\! loop-next { $prefix = “\t|” }
\! loop-last { $suffix = “;” }
            $prefix  test_mode[chip_id]->$name << $offset$suffix
\! end-loop

        case ERX_MCTRP
        …
```

Refactoring the Code Generator

- Consider All alternatives

- If
  - "Standard" refactorings too weak; and
  - "Introduce Code Generator" too powerful
- Then perhaps
  - "Introduce Meta-Data" may be just right

Registers: Code Generator

```
    …
    switch (address)
    {
        case ERX_TestModeReg:
\! loop ($name, $offset) ([“mode2_en”, 1], [“mode1_en”, 0])
\! loop-first { $prefix = “return”; $suffix=“” }
\! loop-next { $prefix = “\t|” }
\! loop-last { $suffix = “;” }
            $prefix  test_mode[chip_id]->$name << $offset$suffix
\! end-loop

        case ERX_MCTRP
        …
```

Introduce Meta-Data

- Internal
  - Describe data using [Perl] constructs
  - Refactor into .pm files (objects)
- External
  - Read from File
    - .txt – record per line
    - .html –for tables (generate from Word?)
    - .xml – for structured data

Registers: Meta-Data

```
@registers = (
        TestModeReg => [
                [ mode2_en => 1 ],
                [ mode1_en => 0 ],
        ],

        MCTRP => [
                ...
        ],

        ...
    );
```

Introduce Meta-Data

- Internal
  - Describe data using [Perl] constructs
  - Refactor into .pm files (objects)
- External
  - Read from File
    - .txt – record per line
    - .html –for tables (generate from Word?)
    - .xml – for structured data

Registers: XML Representation

```
<register name="TestModeReg" address="0010">
   <field name="mode2_en" offset="1">
        <variable struct="test_data" initial_value="0" />
   </field>
   <field name="mode1_en" offset="0">
        <variable struct="test_data" initial_value="1" />
   </field>
</register>
```

Inertia of Code Generators

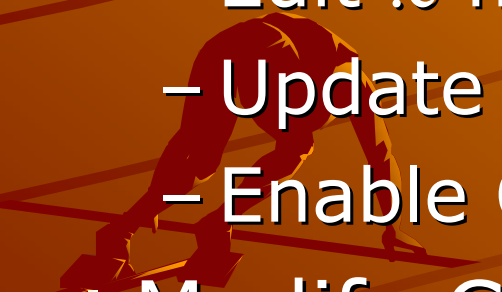
- Generators constrain creativity!
- People avoid changing abstractions
  - violate encapsulation
  - edit generated code
  - contort meta-data
  - change the specification
  - introduce brittleness
- Overuse of Code Generators

Changing Generated Code

- Formatting
- Refactoring (non-meta)
- Add local feature
- Locally disable feature
- Global policy change

Changing Generated Code

- Modify non-meta code
  - Disable Code Generator
  - Add / Modify (non-meta) tests
  - Edit .c files to pass tests
  - Update .gold files with changed .c files
  - Enable Code Generator
- Modify Code Generator (pass tests)
- Refactor Code Generator

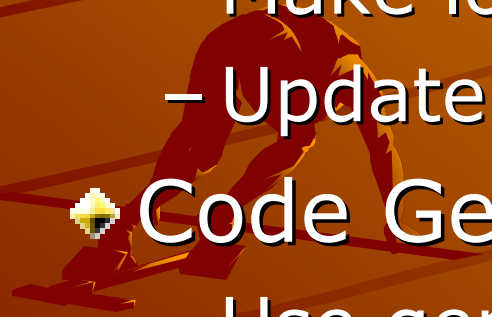


Hierarchy Of Testing

- Acceptance Tests
- Non-Meta Unit Tests
- Code Generator's Acceptance Tests
  - .gold text-diff
- Code Generator's Unit Tests
  - meta-data not directly testable
    - But some validation may be possible
- Repeat

Global Changes

- Always a Refactoring
  - Existing tests should still pass
- Think Globally, Act Locally
  - Make local change manually
  - Update Code Generator
- Code Generator for Global Update
  - Use generated code for new .gold files



Makefile 2

```
DIFF := diff
test: foo.c.diff

foo.c.diff: foo.c.new foo.c
    $(DIFF) $^

foo.c.new: foo.pl
    perl –w $^ > $@
```

The Magnifying Glass

- Be prepared to undo a refactoring
  - But learn from the attempt
- Meta-Refactoring magnifies variation
  - Some variation in necessary
  - Some is not
- Which is which?
  - Eliminate unnecessary variation
  - "Introduce Symmetry"

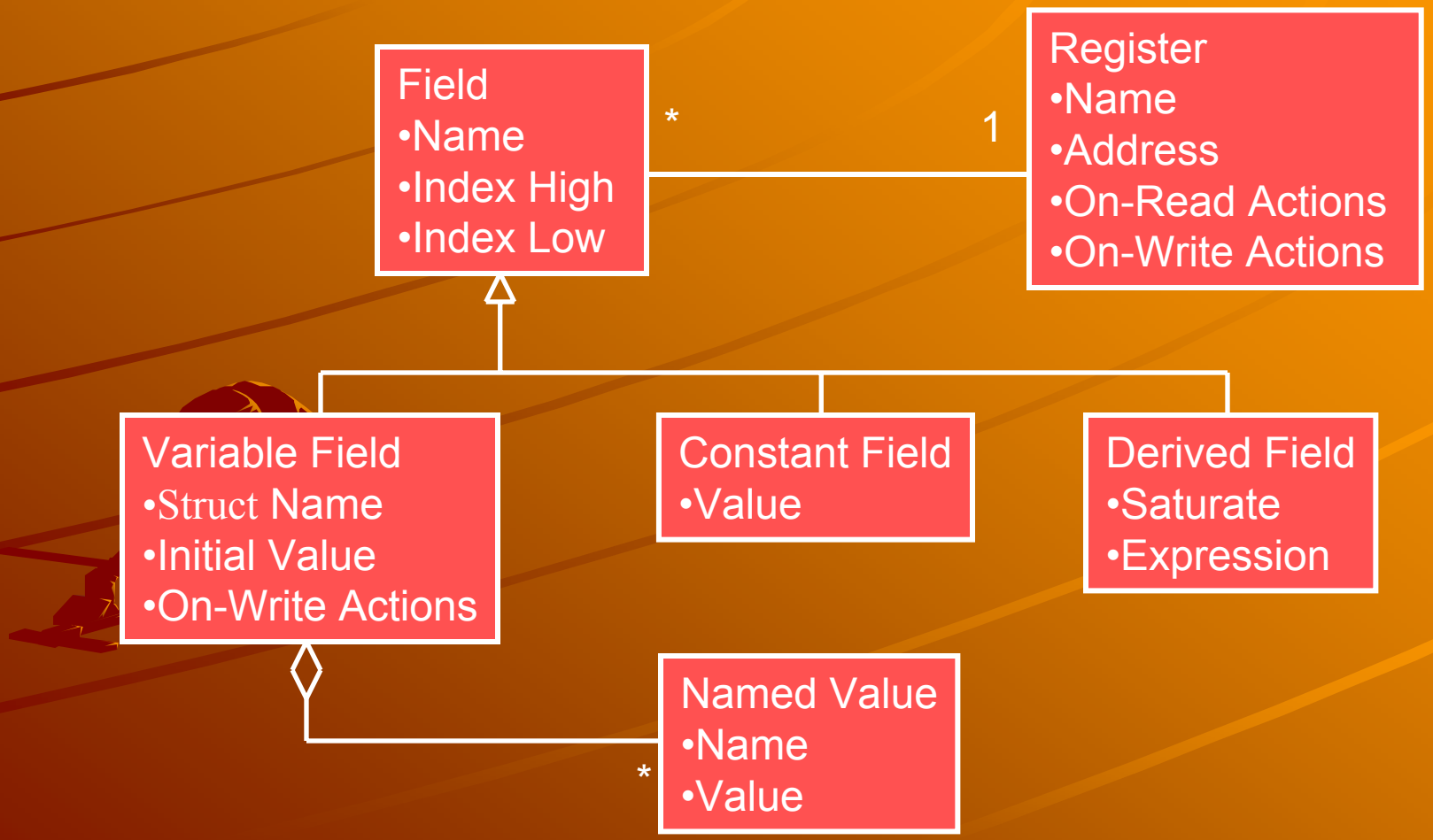
Unification

- Code Generators for multiple files
- Discover Abstractions
- Identify Meta-Data
- Create Meta-Model of Meta-Data
  - Model in UML
  - Refactor Code Generator to use UML

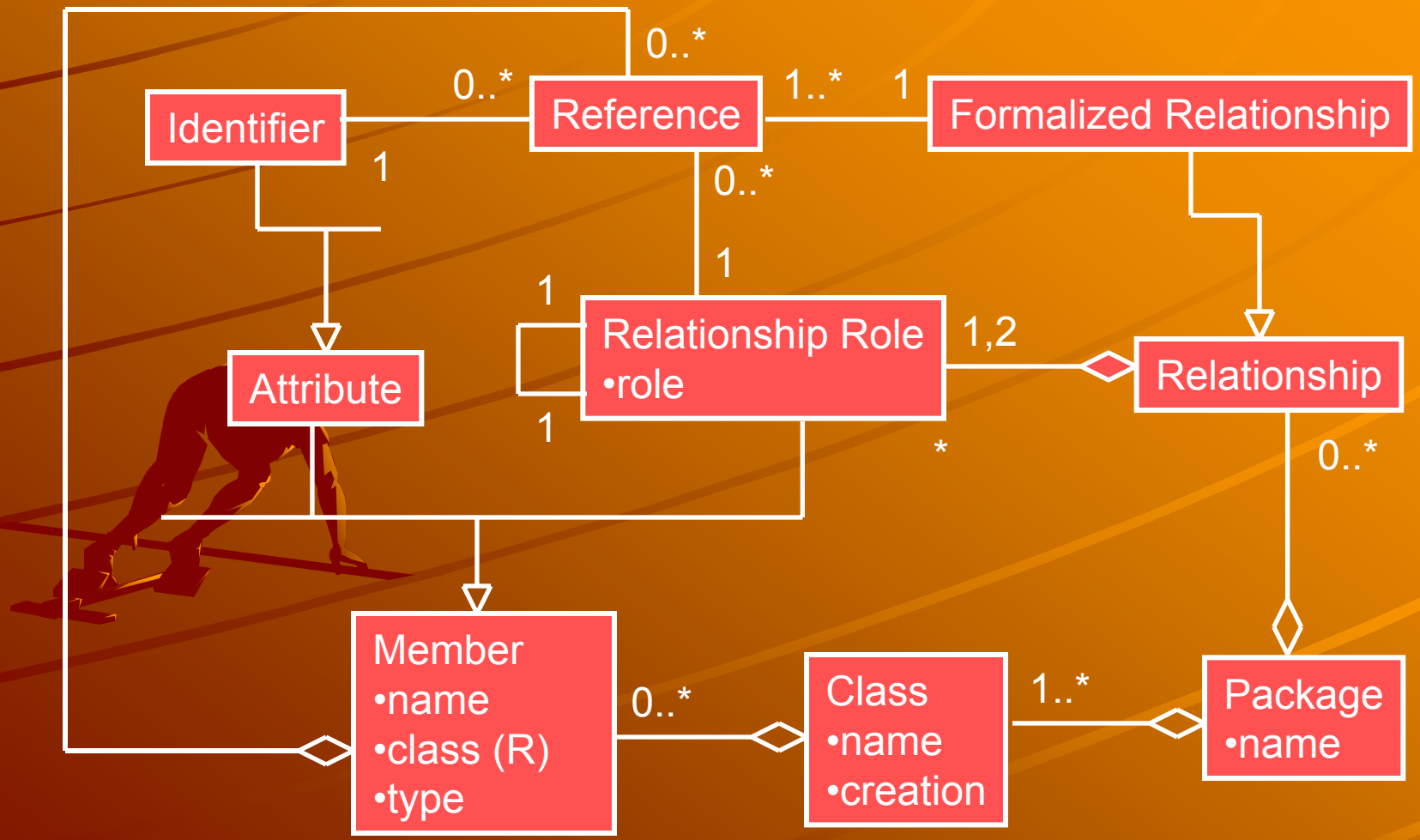
Meta-Model (XML)

```
<package name='Registers_DB'>
   <class name='Register' creation='manual'>
        <identifier name='name' type='string' />
        <attribute name='address' type='hex' />
        <attribute name='on-write action' type='code' />
   </class>
   <relationship
        from.name='Register' to.name='Field'
        from.multiplicity='1' to.multipliciy='*'/>
   …
```

Registers: Code Model



Meta Meta-Model (xUML)



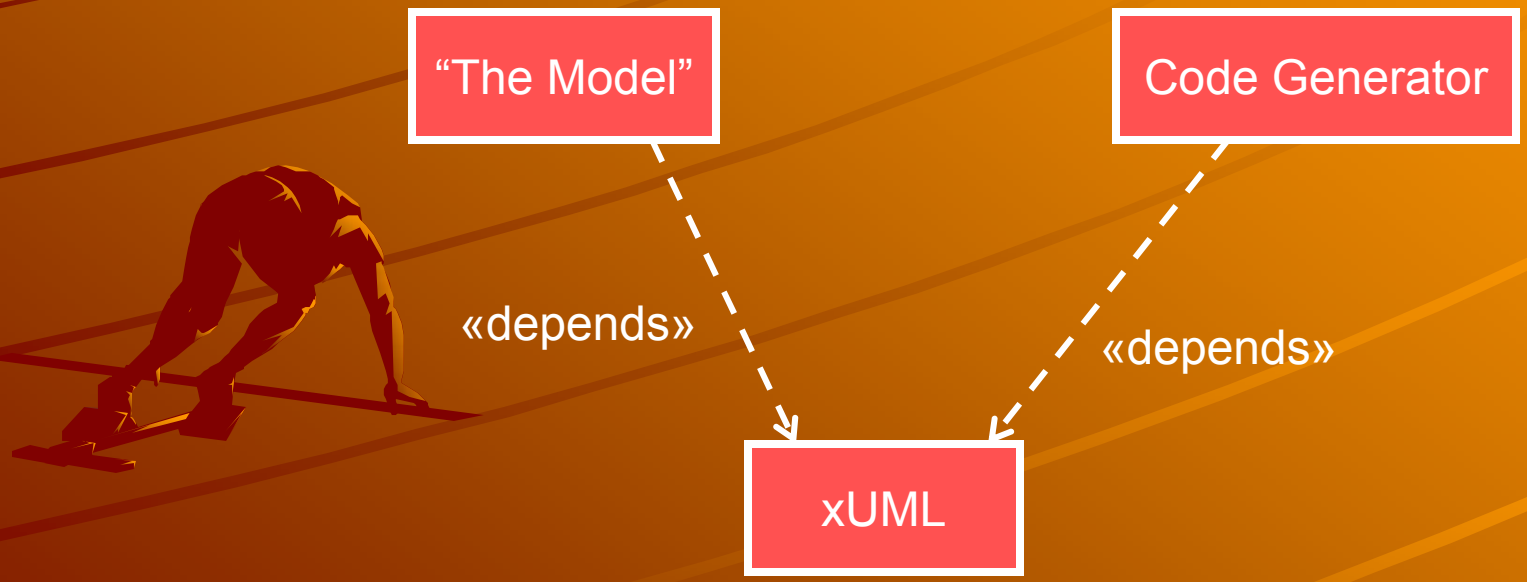
Top-Down Vs Bottom-Up

- Domain Separation
- Bridging
- Meta-Model

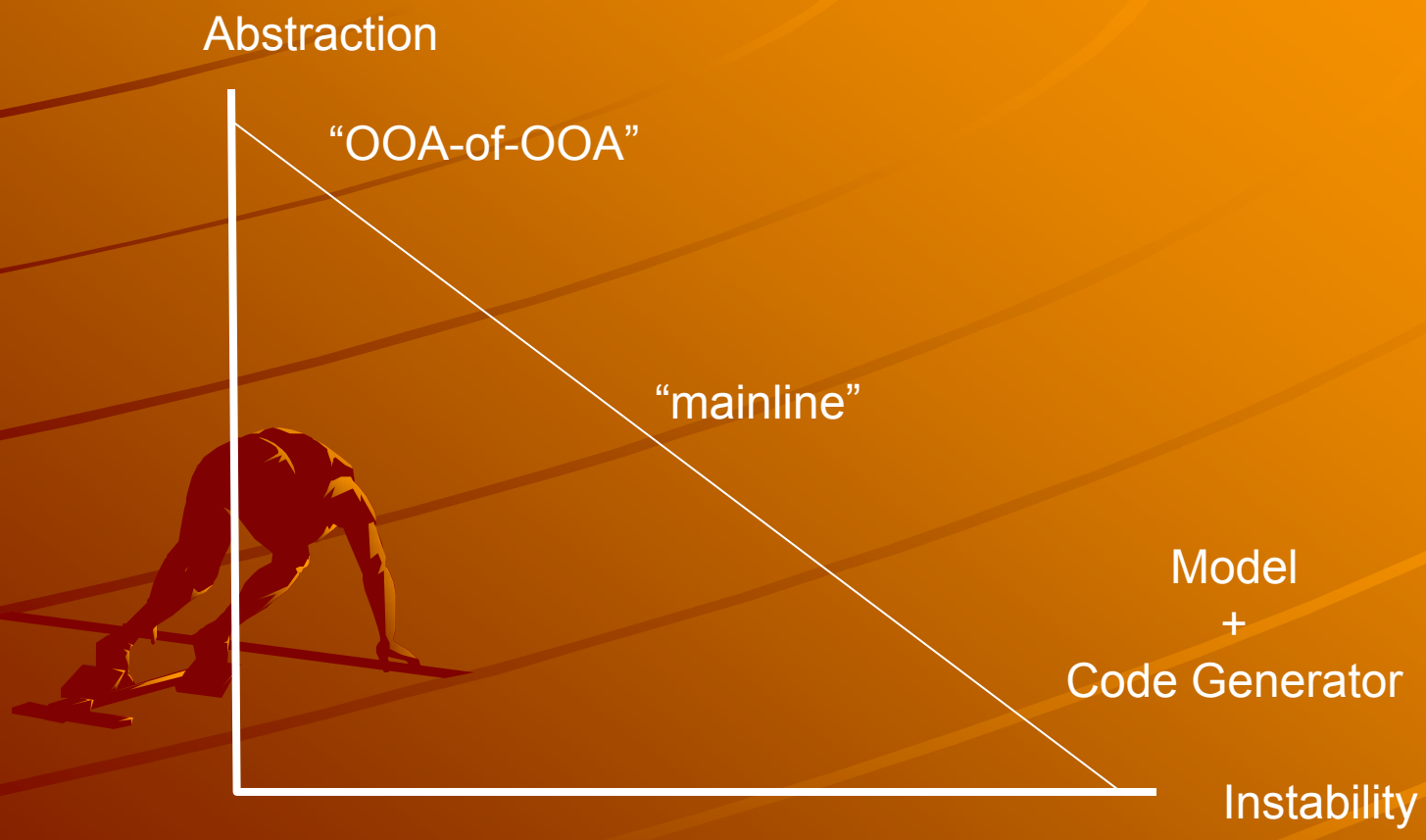
- YAGNI and DTSTTCPW
- Bootstrapping

Modeling

- Top Down Vs Bottom Up



Stability vs. Abstraction



Source: "Stability", Robert Martin, http://www.oma.com

Top-Down Vs Bottom-Up

- Domain Separation
- Bridging
- Meta-Model

- YAGNI and DTSTTCPW
- Bootstrapping

Summary

- Refactorings reveal abstractions
  - Introduce Code Generator
  - Introduce Meta Data
- Code Generators can be Maintainable
  - Don't assume reuse for ROI
  - 100% generation is not necessary
  - Code-Generator as Partner
- Be willing to experiment

Questions

Lunch