

WRITING CODE *that* WRITES CODE

STEVE MARSHALL

Gonna talk about codegen.

Steve Marshall

- Interested in everything
- Want to be a pirate
- Y! Ents

Why codegen

- Don't like repetition
- Don't like repetition
- Interesting problem of scale
- Lack of resources => be creative

But why?

Quality, consistency, single point of knowledge, eases bulk refactoring, improves architectural consistency, increases abstraction, increases morale.

Quality

Lots of code => inconsistencies
Templates => consistent quality

Consistency

- Consistent naming
- API style
- No surprises

Single Point of Knowledge

Schema describes system simply
Architecture documented, not embedded in code

Facilitates Bulk Refactoring

Faulty assumptions in hand coding mean refactoring lots of code
Generation makes refactoring minimal
Also: generation can compress project timelines => more time to design/prototype, less need to refactor

Architectural Consistency

- Generator encourages working within architecture
- "generator doesn't do that" can indicate that the feature doesn't fit the architecture
- Consistent, structured approach, regardless of team members joining/leaving

Abstraction

- Application logic is language-independent. Can easily port templates to other languages (eg. for performance, also: WSDL describing inputs/outputs)
- Can easily review/validate abstract designs
- Aids development of non-implementation code (tests, documentation, etc)

Full domain language

Tier generation

Partial class generation

Mixed code generation

Inline code expansion

Code munging

Type of codegen...

- Munging
- Code expansion
- Mixed codegen
- Partial class gen
- Tier gen
- FDL

Full domain language

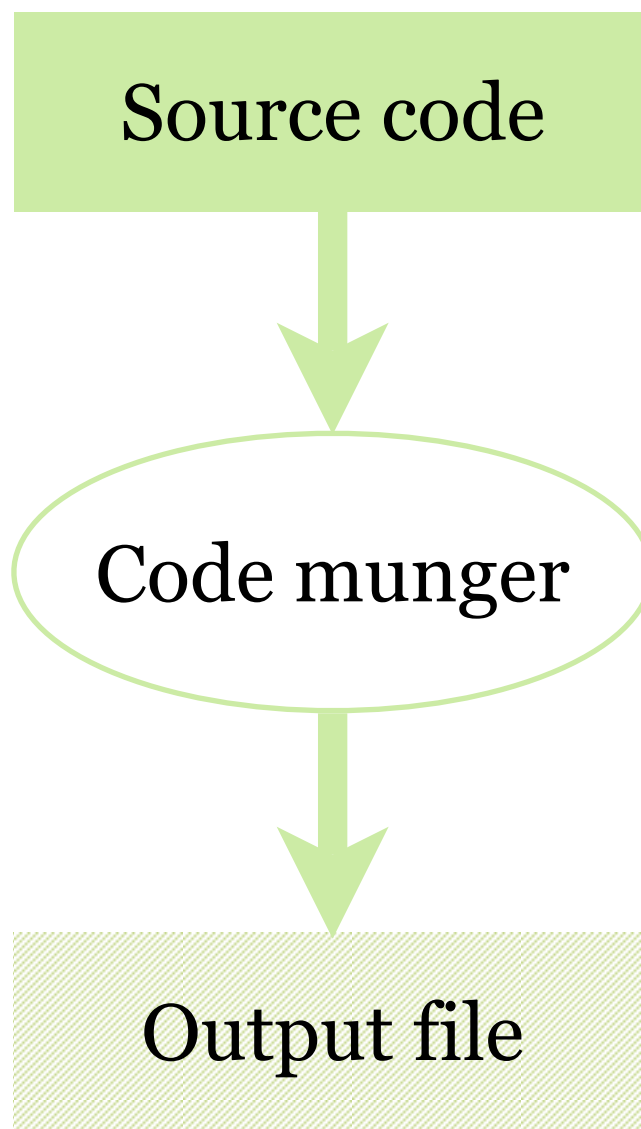
Tier generation

Partial class generation

Mixed code generation

Inline code expansion

Code munging



Parses source code
Produces some non-source output

See documentation, linting, ... Also: build header files, web service layers

Full domain language

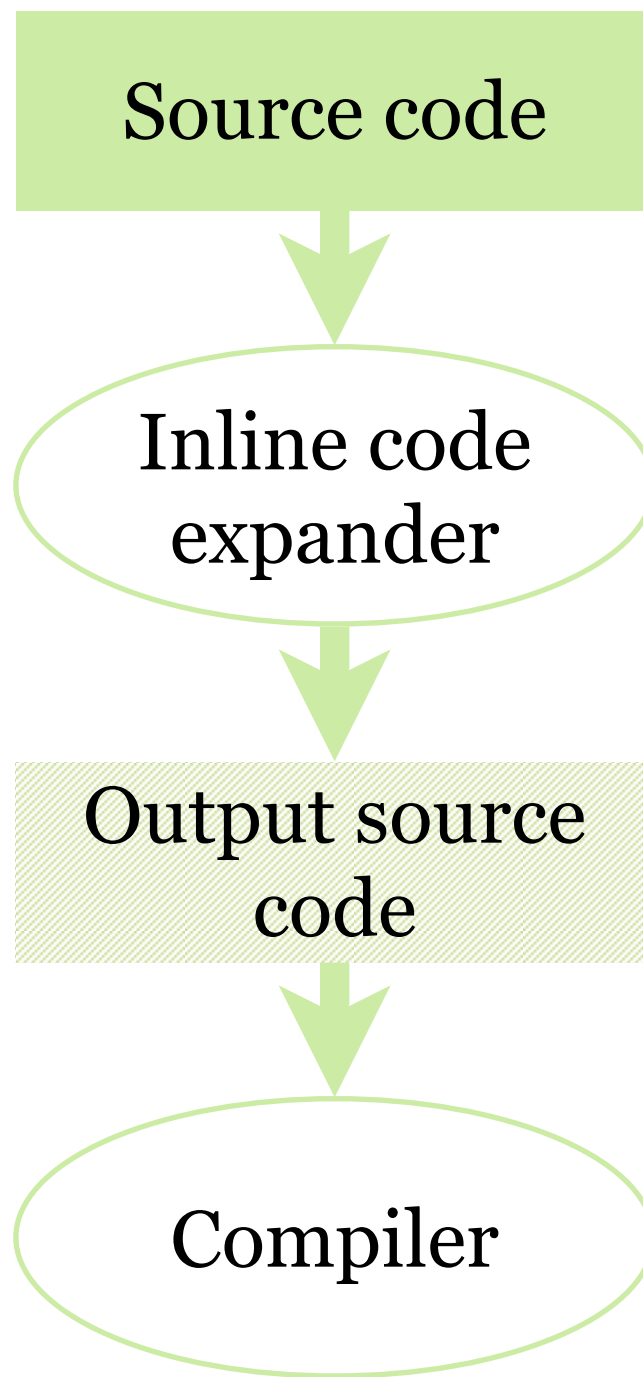
Tier generation

Partial class generation

Mixed code generation

Inline code expansion

Code munging



Parses source for special keywords
Replaces with production code

eg. Embedding SQL in source à la Pro*C/SQLJ

Full domain language

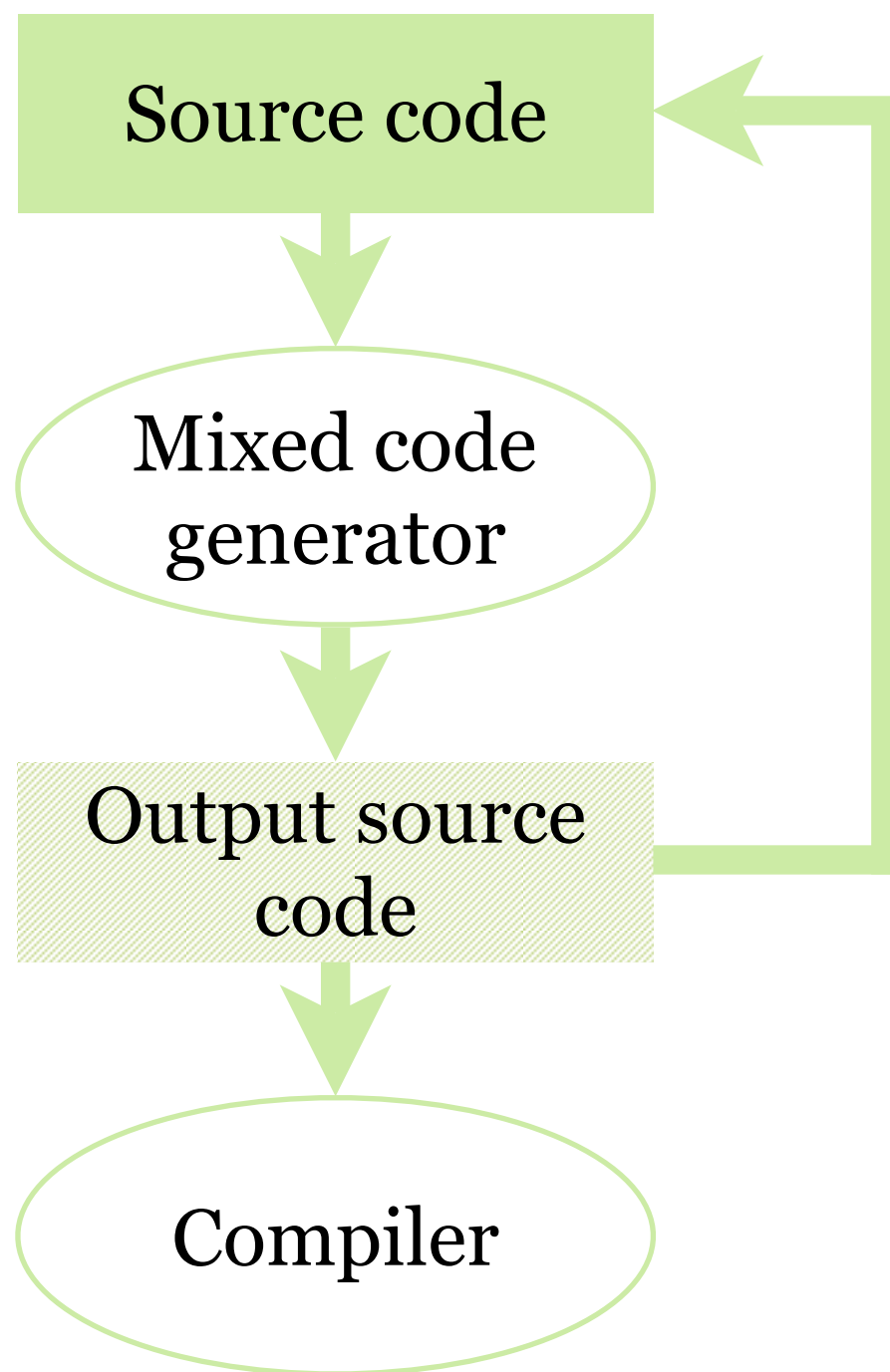
Tier generation

Partial class generation

Mixed code generation

Inline code expansion

Code munging



Like inline code expansion, but output can be used as input (**replaces in-place**)

Looks for specially formatted comments, fills comment-wrapped area with new production source
eg. Marshalling code

Full domain language

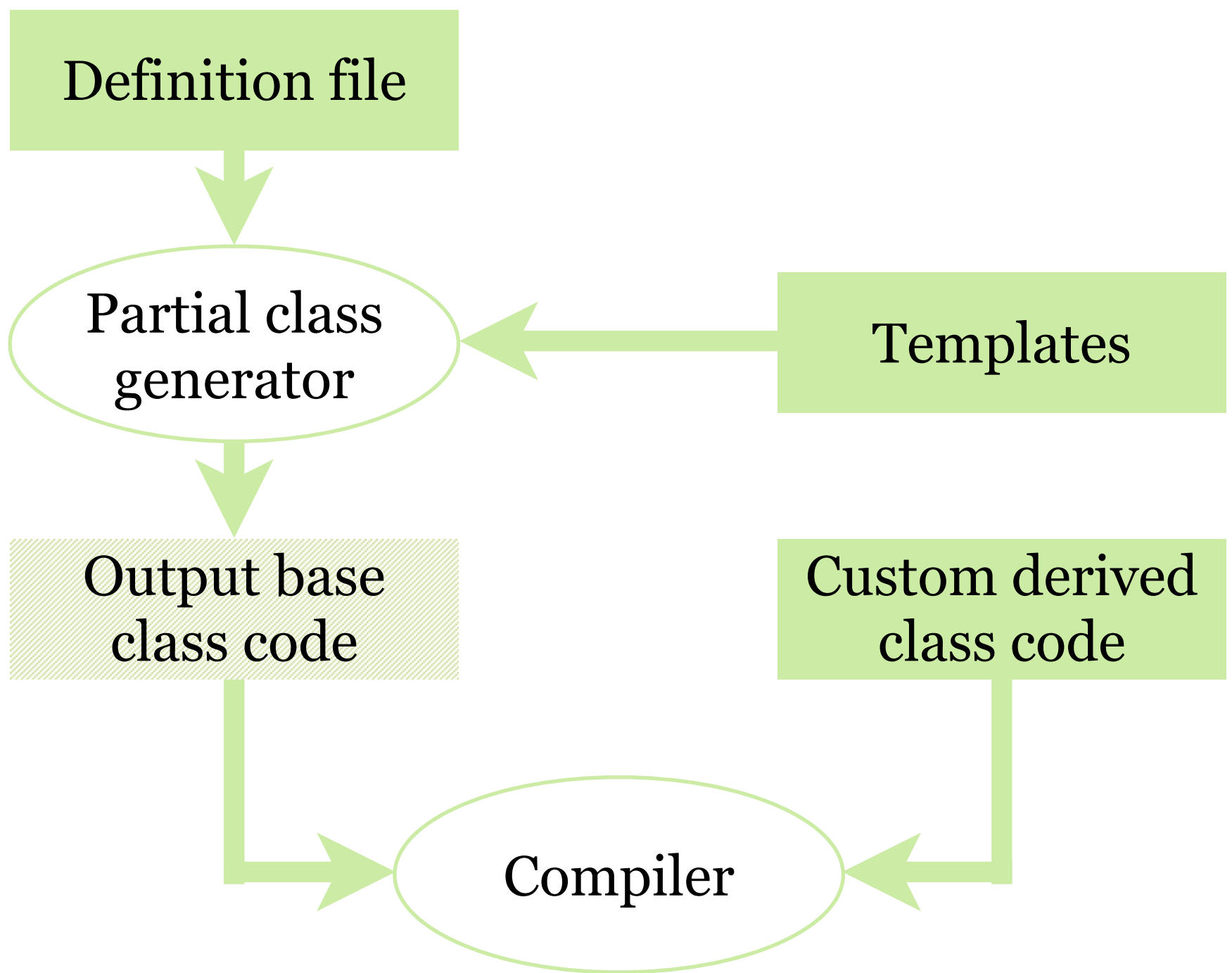
Tier generation

Partial class generation

Mixed code generation

Inline code expansion

Code munging



Reads **abstract definition** with enough info to build classes
Applies def'n to templates to output base classes
Used with **subclasses for custom behaviour**

Can be a stepping stone to **tier generation**

Full domain language

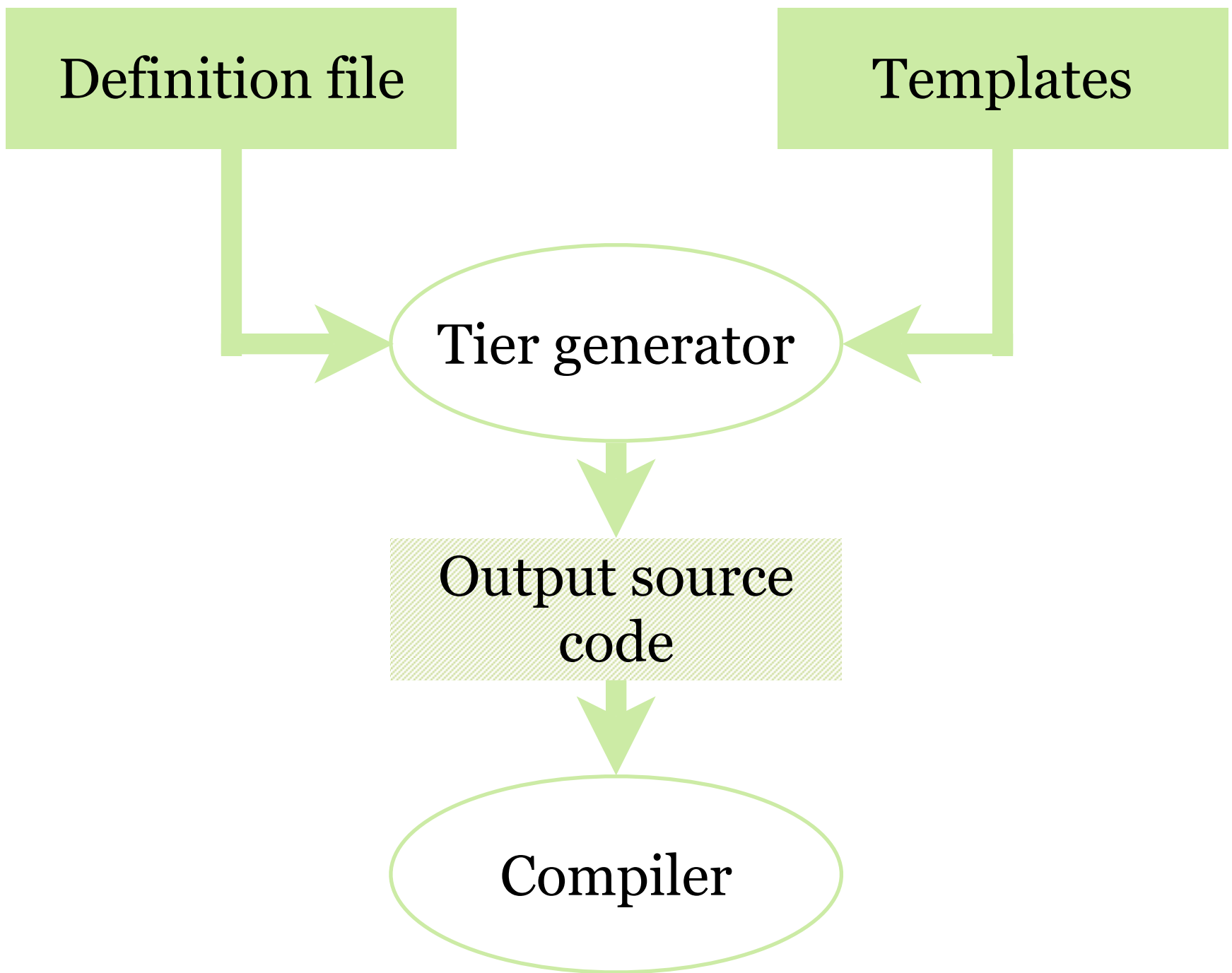
Tier generation

Partial class generation

Mixed code generation

Inline code expansion

Code munging



Generator builds a complete tier of your app (eg ORM)
eg. Model-driven generation takes definition file and applies to templates

Like partial class, but without special cases (or are codified in templates)
Partial class generation requires extra code, tier doesn't (but may have it)

Full domain language

Tier generation

Partial class generation

Mixed code generation

Inline code expansion

Code munging

Turing-complete language mapped to your problem space.
Expressive and powerful, but... [\[click\]](#)



You are not Stephen Wolfram.

Even if you think you need a full-domain language, you probably don't. Not right now, anyway.

Lots of work.
Requires training everyone on it.

Not going to talk about full-domain languages.

What should I use?

Off-the-shelf tools:

- r3 FTW!

Build your own:

- Simple to start
- Scripting language: Perl, Ruby, Python, ...
- Templating language: TT2, Jinja, ERb, ...
- Definitions: XML, YAML, JSON, ...

Let me tell you a story...

Waffle about TV listings:

- 40+ object types
- Similar requirements per type
- Generate!
- Saved hundreds of man-hours
- Now have a tool that can generate a PHP ORM for any data model
- Looking to open-source

One for **the team.**

The first step is admitting you have a problem.
Do you even need codegen?

Code generation is good for repetitive tasks, particularly in large projects

Formal suggestion

Fully fledged project

Follow normal project process

How does the generator fit into your development lifecycle?

Skunkworks

Build it yourself

Introduce it quietly to others

Eventually tends towards formal project

“Just for me” tool

Like skunkworks, but without the evangelism

May cause perception problems

1.

Respect (and loathe) hand-coding.

Respect

Often need special cases: accommodate these

Loathe

Time is valuable, wasting time on repetitive tasks is criminal
Generator should optimise creativity and enthusiasm of engineers

2.

Hand-code **first**.

Must fully understand your intent before generating.
Handwrite a significant spectrum of code and use that as basis for templates.

3.

Control your source.

Use. Source. Control.

Seriously.

If your generator works on implementation files with hand-written code, need to ensure that's protected.

Control your source.

4.

Consider the
generator's language.

Don't have to use same tools to generate as to write the resulting app
Different problem-space, pick tools accordingly

5.

Integrate the generator into your workflow.

- Tool to be used by engineers.
- Should fit cleanly into process
- If they use an IDE, try to integrate there.
- Consider commit-hooks and build scripts

6.

Include warnings.

- Warn people not to hand-modify gen'd code
- If they do anyway, /help/, don't berate

7.

Play nice with others.

- For engineers, not automatons
- Should tell what it's doing, where, and handle errors reasonably
- If it's difficult or flaky, it'll be ignored

8.

Document well.

- Selling point for generator
- Thorough, but not overwhelming
- Highlight key points: what it does, how to install, how to run, what it affects

9.

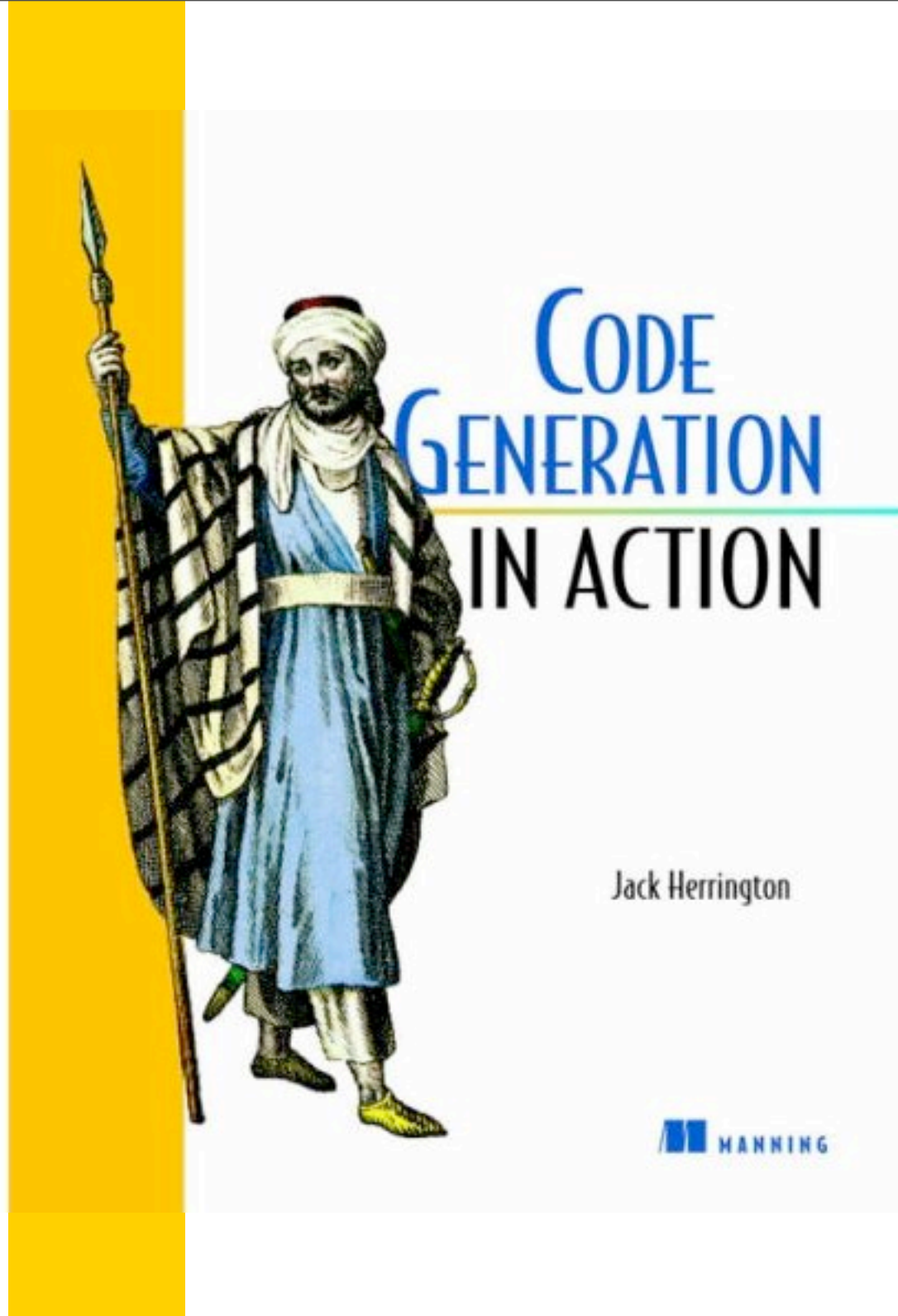
Consider the cultural impact.

- Educate colleagues through one-on-one meetings, documentation, seminars, etc.
- People are skeptical of new things: break through this and emphasize that you designed the generator for their benefit

10.

Keep it **maintained.**

- Unless it's a temporary measure, maintain it
- Budget for updating and maintaining it, like any other tool



Everything I know, I learned from Bill Hails and Jack Herrington.

Bill doesn't have a book on code generation, but Jack does. This is it. Buy it.

icanhaz.com/codegen-book
[/codegen](http://icanhaz.com/codegen)

@stevemarshall
steve@nascentguruism.com

Stephen Wolfram photo:
flickr.com/photos/hybernaut/87907765/

The book.

These slides.

Me.