ACCU 2009

legacy code

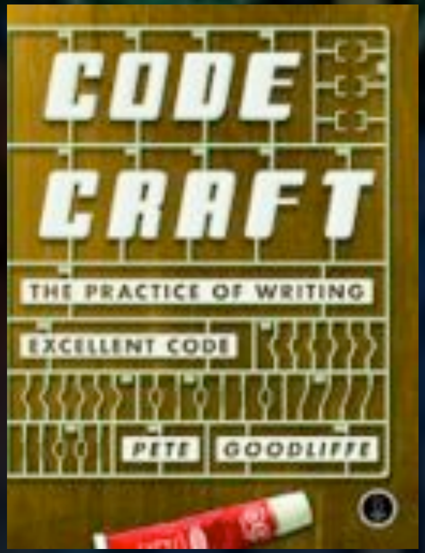
learning to live with it

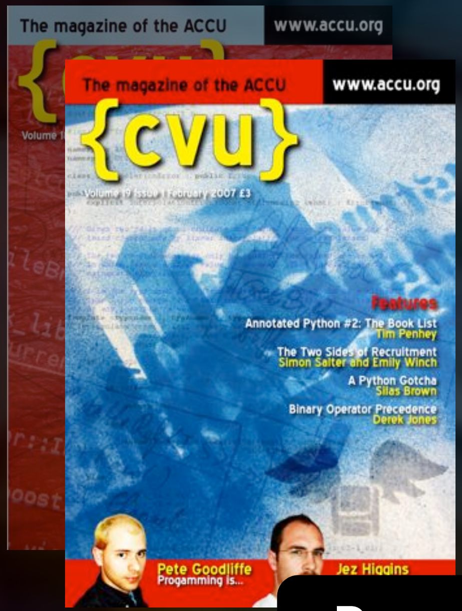
Pete Goodliffe
pete@goodliffe.net

Pete Goodliffe

A programmer, a columnist, an author, a teacher. Someone who cares about code. Books: Code Craft, Beautiful Architecture.

www.goodliffe.net
goodliffe.blogspot.com





Pete Goodliffe
pete@goodliffe.net

Eve
Adam

most software is

most software is

Calvin Klein
Nintendo
BIGGER IS BETTER
JOHN DEERE
Calvin Klein

talk synopsis

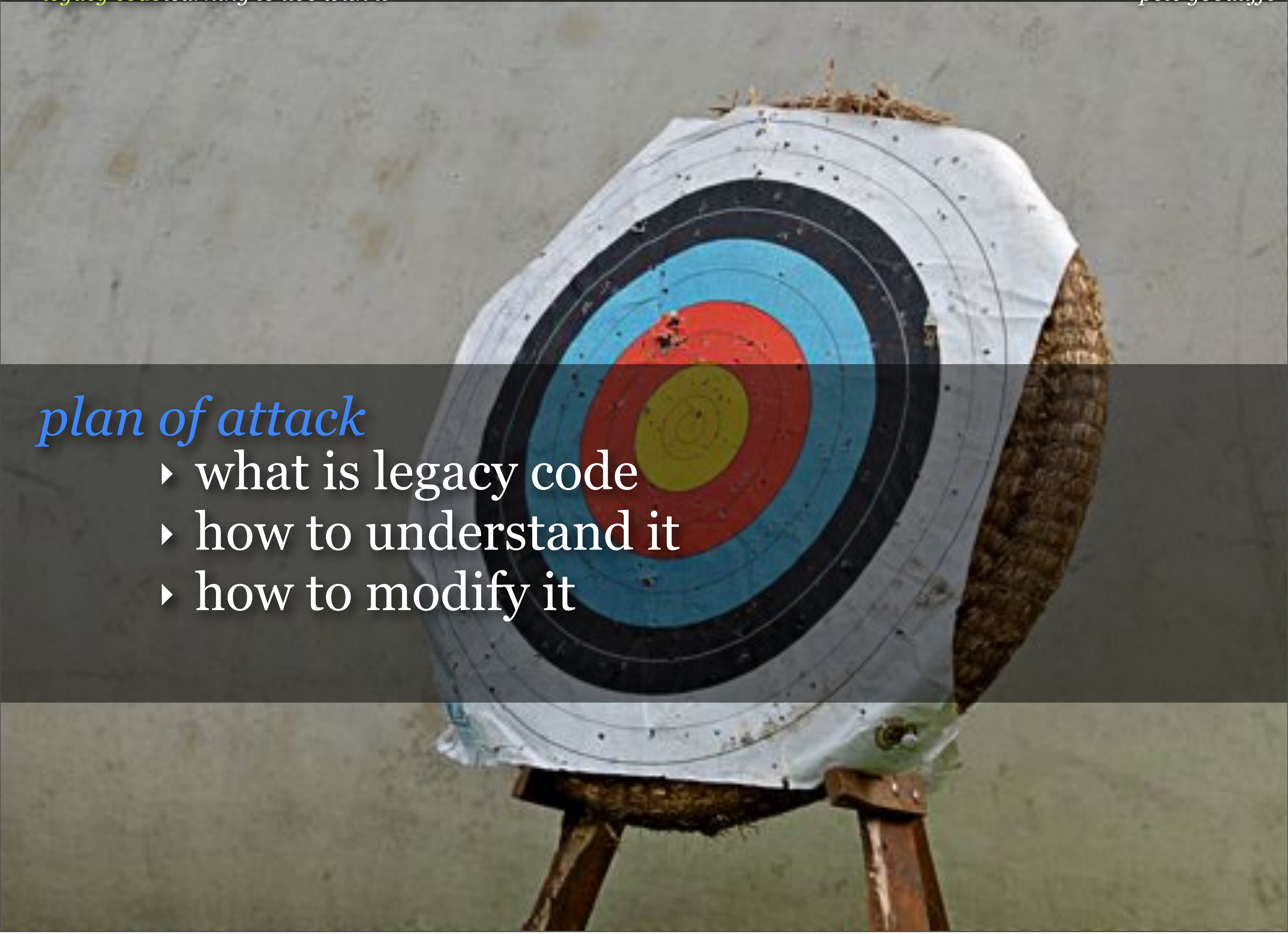
Legacy code. You can't live with it. You can't live without it.

Well, you can't avoid it, at least. Spend long enough in the software factory, and you'll inevitably run into other people's old code. And of course, none of this old stuff is any good. It's nothing like the high quality software you craft. Pure tripe.

Let's be honest, sometimes you might even stumble across some of ***your own*** *old code, and embarrassing as it is, you have to admit that you don't know how it works, let alone how to fix it.*

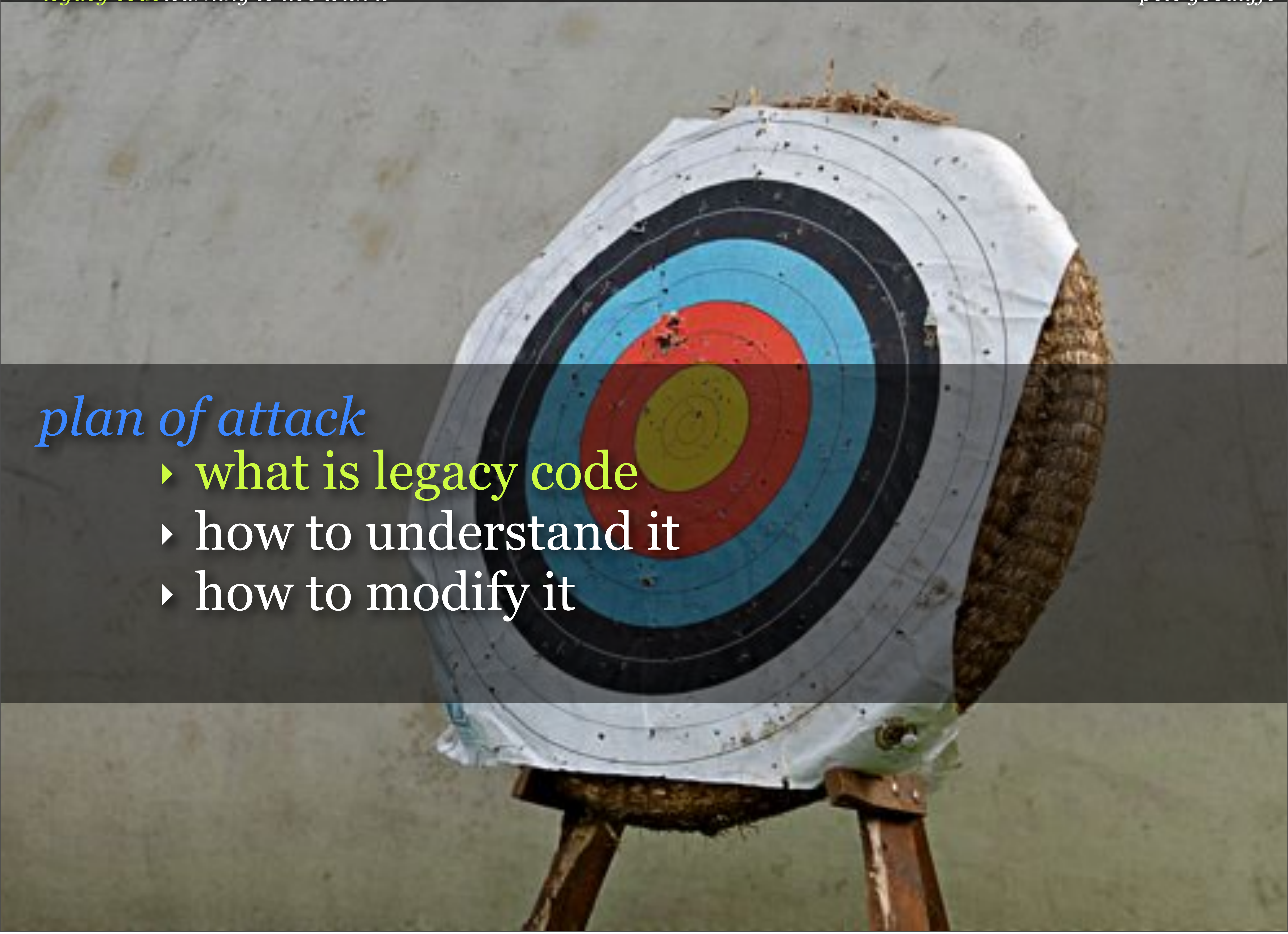
This presentation will look at practical strategies for working with "old" crufty code. We'll see how to:

- ***start working*** *with a completely unfamiliar codebase*
- ***understand*** *old spaghetti programming*
- *make correct* ***modifications***
- *prevent bad code from causing more pain in the* ***future***



plan of attack

- what is legacy code
- how to understand it
- how to modify it



plan of attack

- what is legacy code
- how to understand it
- how to modify it

legacy *(noun)*

1. *Law*. a gift of property, esp. personal property, as money, by will; a bequest.
2. anything handed down from the past, as from an ancestor or predecessor: the legacy of ancient Rome.

what is legacy code?

- *Old* code
- Any *existing* code
- *Out-of-date* code
- Code *you didn't write*
- No longer *supported* by supplier
- From a previous product *version*
- Code without *tests*
- Uses *old technology*
- *"Bad"* code

There is a *lot* of legacy code being written right now

why do we care?

- Requirements change
  Old code needs to be extended
- Bugs are discovered
  Old code needs to be fixed
- Technology changes
  Old code needs to be ported

is it actually bad?

not *necessarily* *

* terms and conditions apply

who works with it?

muggins here

(good luck with that)

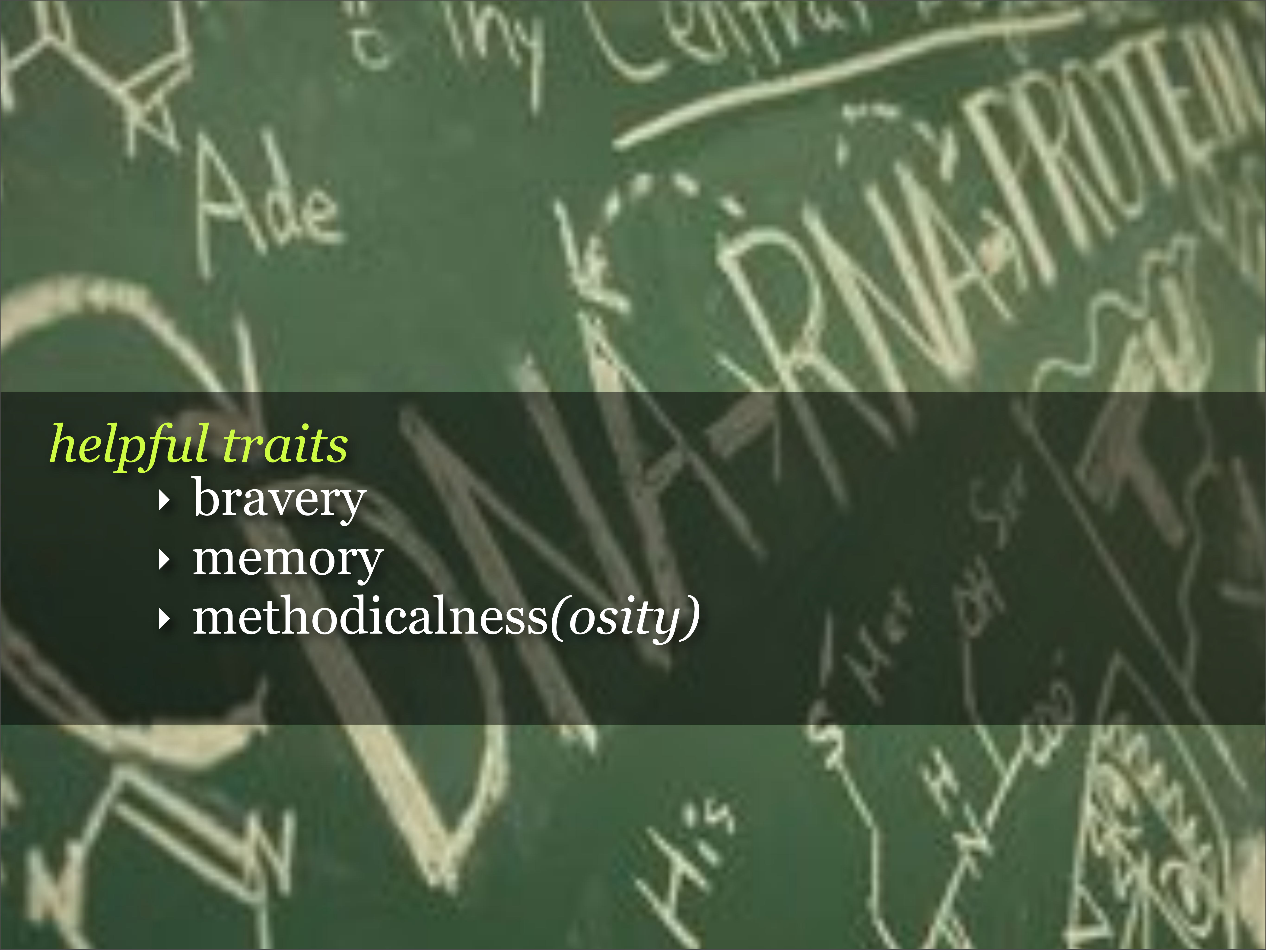
skills >>

helpful traits

- bravery

helpful traits

- bravery
- memory

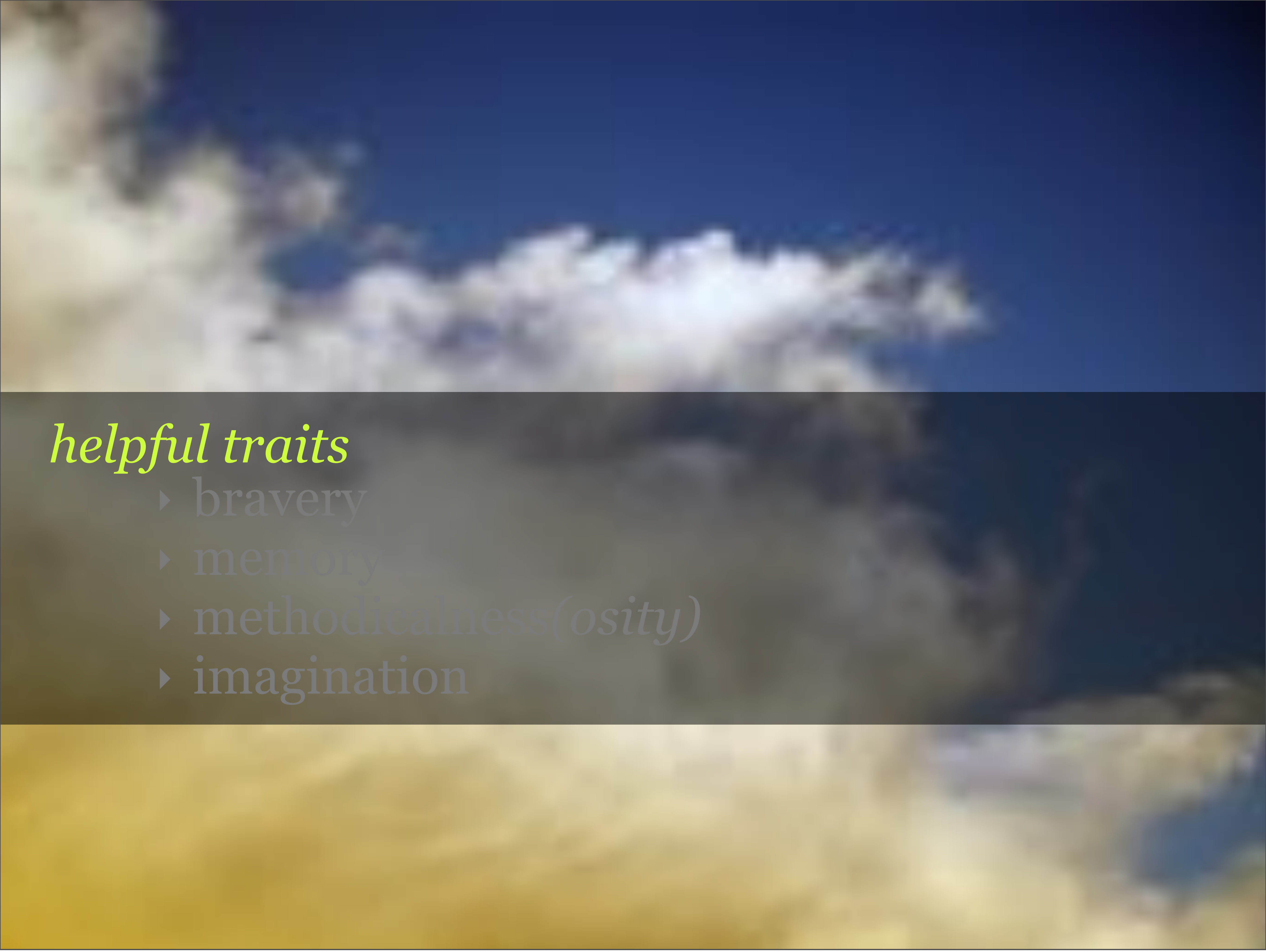


helpful traits

- bravery
- memory
- methodicalness*(osity)*

helpful traits

- bravery
- memory
- methodicalness*(osity)*
- imagination

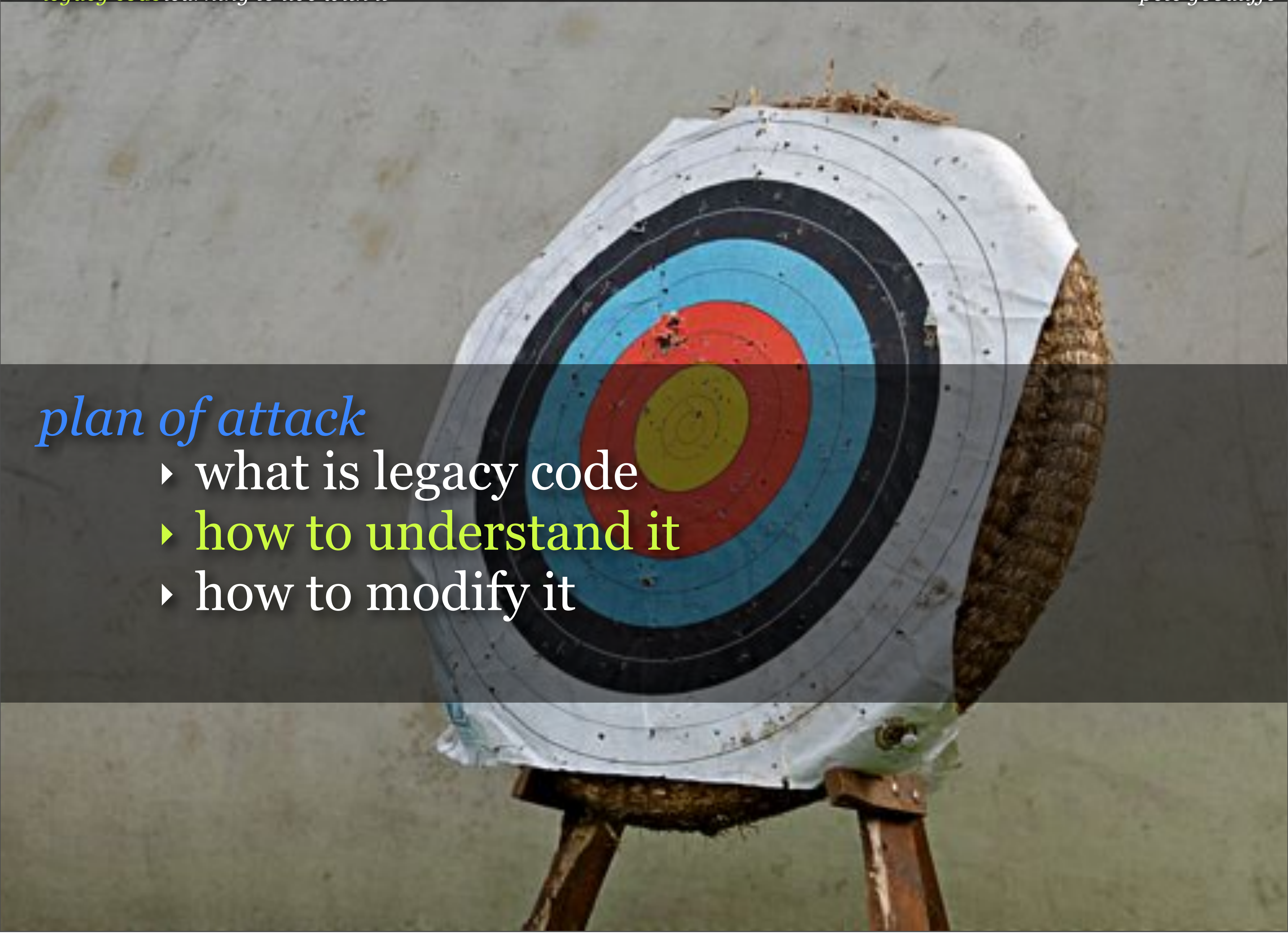


helpful traits

- bravery
- memory
- methodicalness*(osity)*
- imagination

helpful traits

- bravery
- memory
- methodicalness*(osity)*
- imagination
- patience
- intelligence
- empathy
- experience
- persistence
- curiosity
- application
- dedication



plan of attack

- what is legacy code
- how to understand it
- how to modify it

<< understanding

modifying >>

Everything that irritates us about others can lead us to a better understanding of ourselves.

Carl Jung (1875 - 1961)

you have to understand

- the software you are changing
- the changes you must make
- the code you are changing
- how to approach the code

understand: the software

- what *type* of software is it?
  - *e.g. shrinkwrap, server, bespoke*
- what does it do?
- what does it do? *really*?
- have you used it?
- how is it tested?
  - *what QA is there?*
- is there documentation?
- are there manuals?
- gauge the quality (e.g. bug count, reliability)

understand: the software

(people)

- who has domain expertise?
  - *do you need domain expertise?*
- who wrote it?
- who owns it?
- what’s the license?
- who are the users?
  - *are they technical?*
  - *have they been involved in development?*

understand: the software

- what platform(s) does it run on?
- how is it deployed?
- what dev processes is it encumbered by?

understand: the software

- where is it stored?
- change control
  - where is the repository (what system)
  - trunk/branching strategy
    - feature/release/personal branching
  - who can commit, when
    - who else is working on the same branch as you?
    - can you break build?

understand: the software

- other procedural tools
  - bug tracker?
    - bug management process?
    - who manages?
    - who hands out bugs?
    - who gives you an account?
  - continuous integration
  - testing process
    - how thorough?
    - is it automated?

understand: your approach

the right attitude

Weakness of attitude becomes weakness of character.

Albert Einstein

- don’t freak out!
  - someone once understood it
- conquer disgust
- you can improve it

understand: your approach

strategise

become effective by being selective

- how much time do you have to work with it?
  - *affects how you work a route through it*
- how long will you be working with it for?
- how much of it do you need to know?

understand: the changes

what do you have to do?

Do not, for one repulse, forego the purpose that you resolved to effect.

William Shakespeare, 'The Tempest'

- what was the *old* behaviour?
- what will the *new* behaviour be?
- how will you know you are done?

understand: the changes

what do you have to do?

- is it a single coding task?
- or ongoing work in the system?
  - *drive-by programming?!*
- will you take responsibility for whole section of code?
- are you on a schedule?
  - *do you agree with work packages?*

understand: the code

this is the real task: *mapping the software*

- the usual approach: *guesswork*
- a better approach: *structured investigation*

understand: the code

this is the real task: *mapping the software*

- the usual approach: *guesswork*
- a better approach: *structured investigation*

understand: the code

#1: the basic facts

- the language(s)
  - and the language version (e.g. C# 2.0, C89, Python 2.0)
- the size
  - LOC, classes, files, age (does this seem in keeping with project?)
- the build technology
  - check every build variant
- how its deployed
- main technologies
  - libraries
  - database(s)?
  - design tools
  - validation/QA tools
  - external dependencies

understand: the code

build it. now.

- don’t go any further until you’ve got it cleanly built and running
- only then can you modify anything sanely

understand: the code

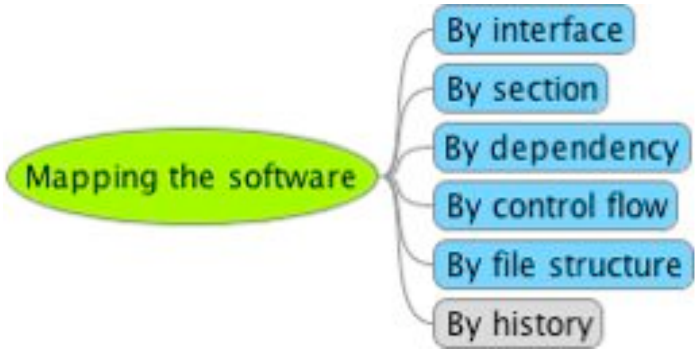
(the map)

find your route in

- is the code structure
  - data-centric
  - control-centric
- does the system decompose into parts?
  - for separate build
  - for separate use
  - which bits do you need to look at now?
- can you ask someone?

understand: the code

find your route in (the map)



Mapping the software

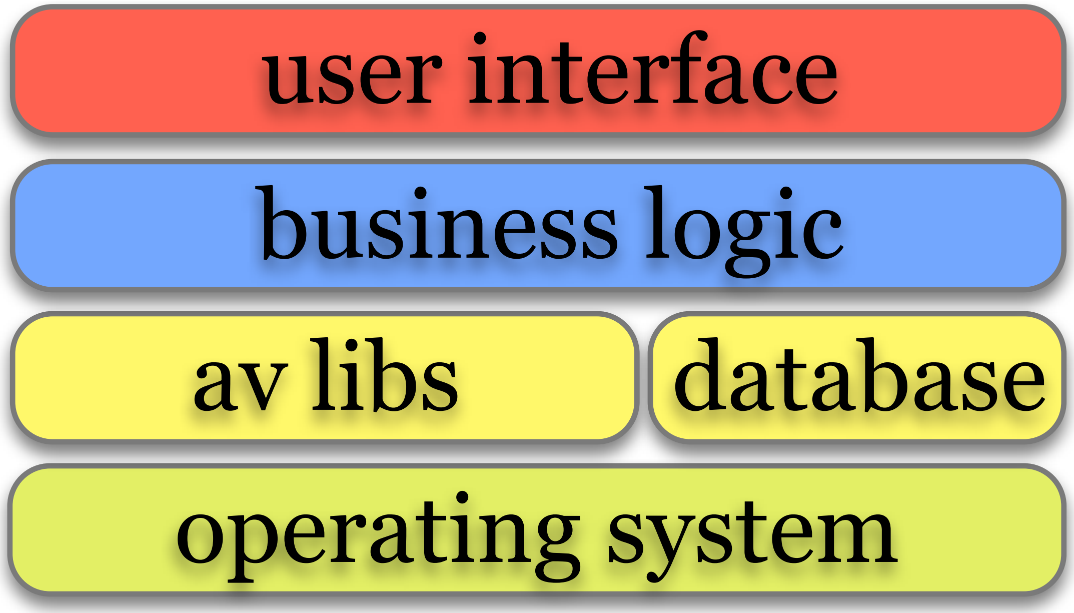
- By interface
- By section
- By dependency
- By control flow
- By file structure
- By history

as you find a route

gauge the quality >>

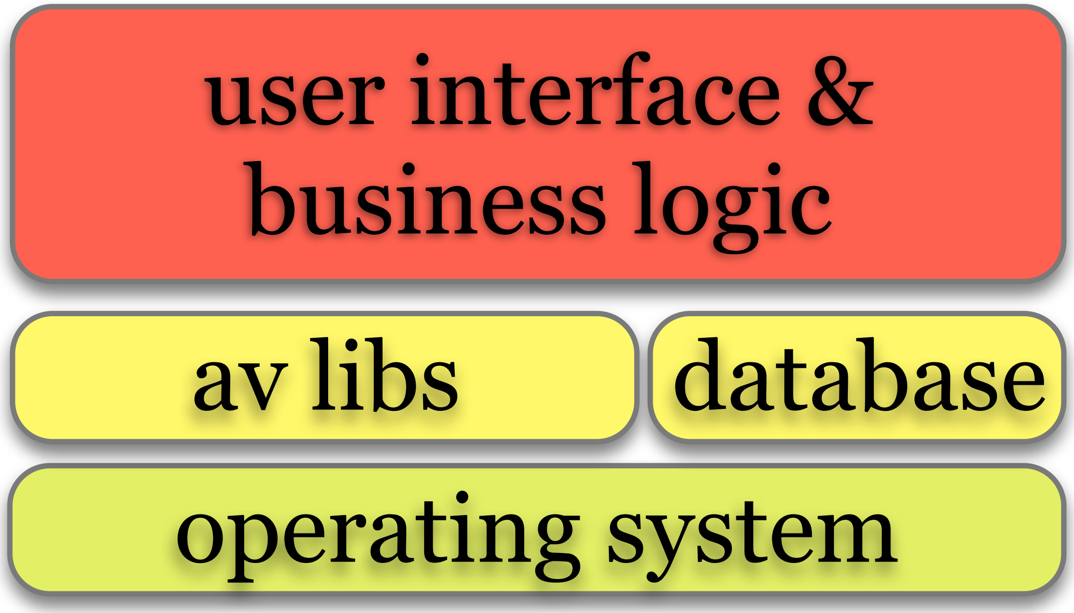
mapping by guesswork

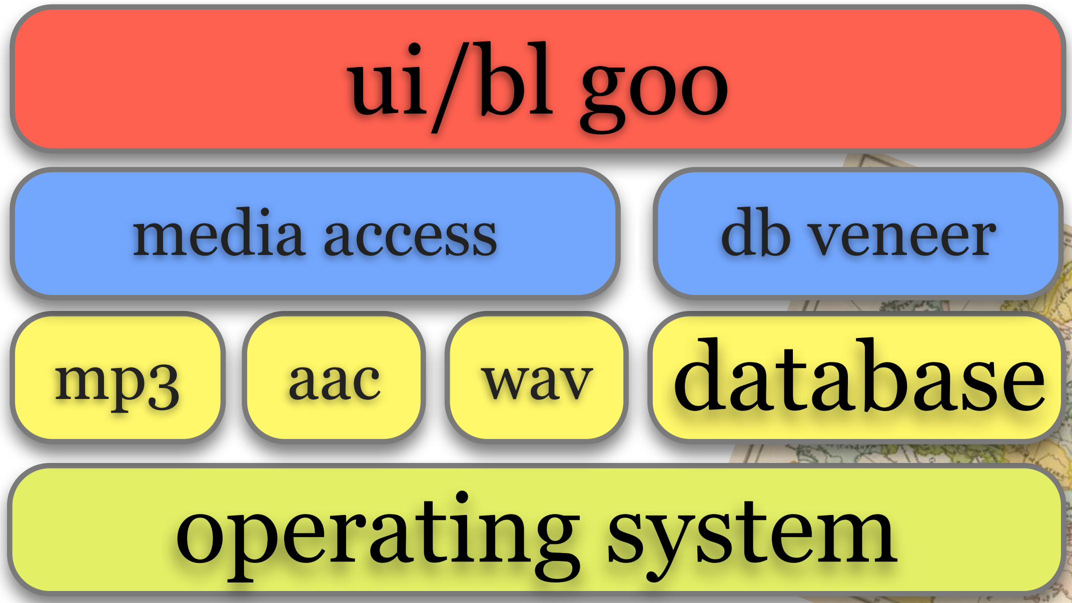
- the first resort
- what do *you* think it should look like?
- what subsections do you expect to find?
- *build a mental model: your map*



mapping by interface

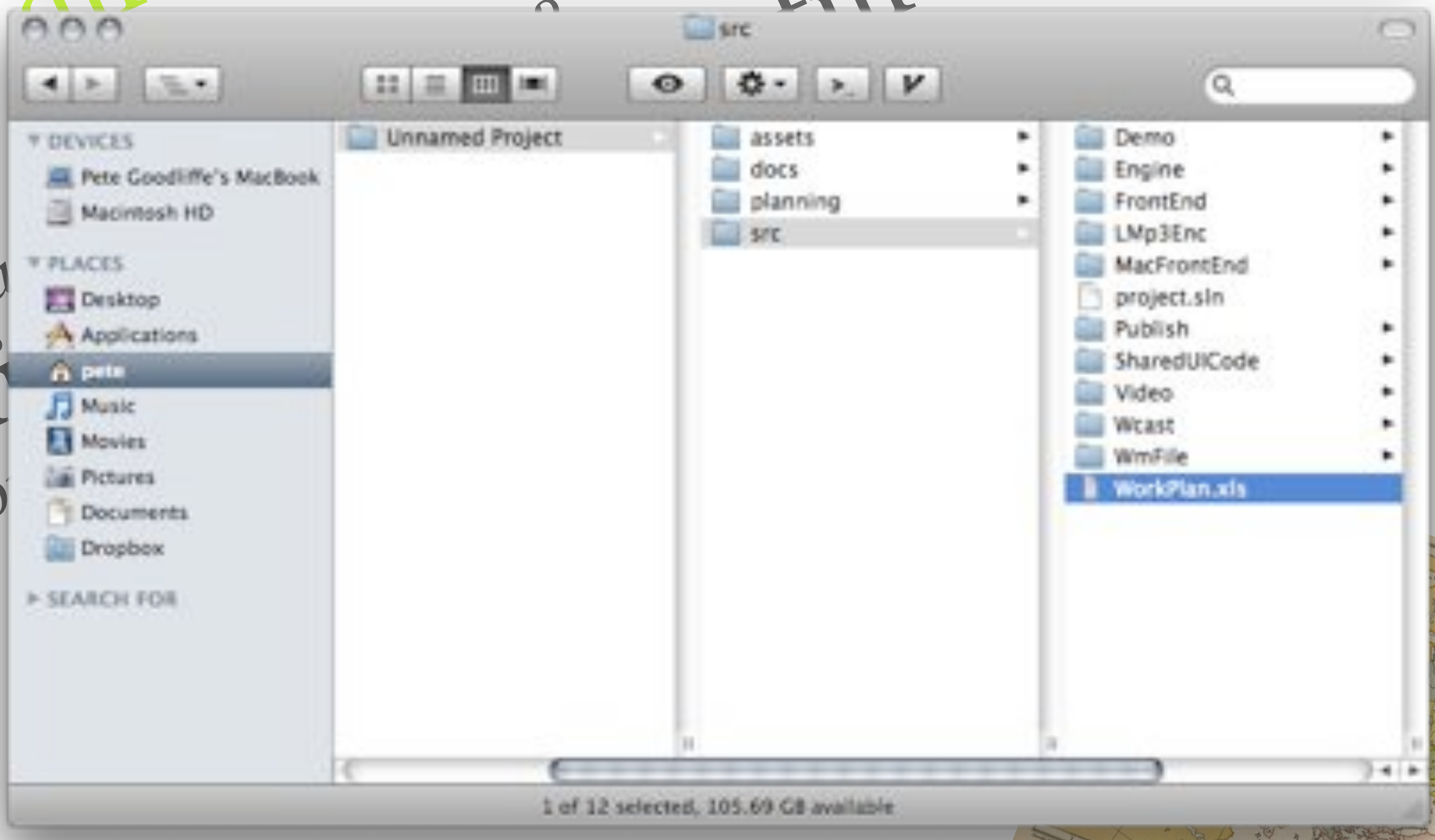
- identify interface points
  - the places in system where subsystems interface
- the nature of the interfaces
  - technology, style, quality, breadth
- high-level / low-level
- *refine your map*





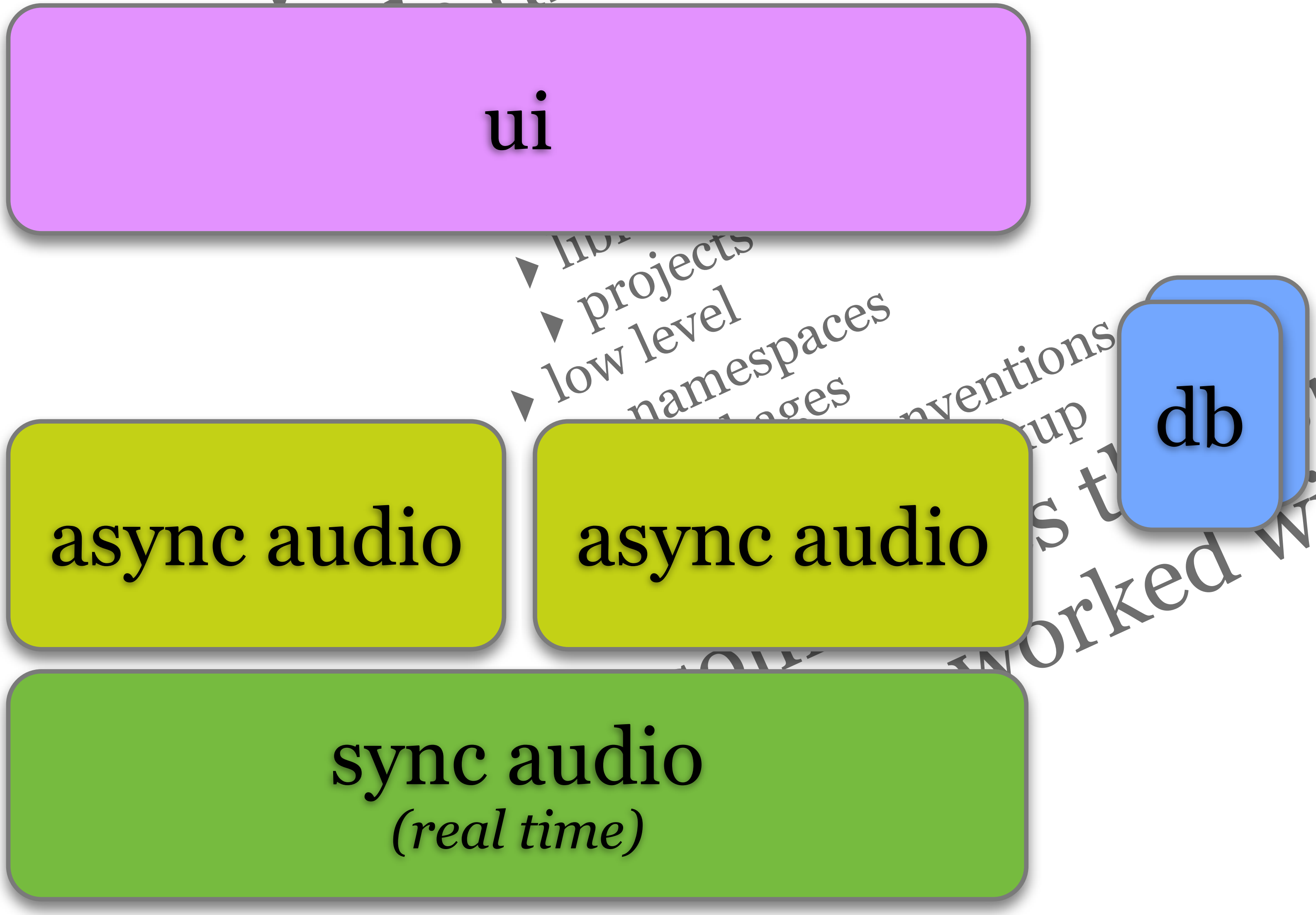
mapping by file structure

- can give a valuable insight
  - either shows internal structure of project
  - or lack of internal structure of project
  - clues for quality of project
- the process:
  1. find the code directory structure
  2. plot the tures
  3. QED
     - does it ma
- recogni
  - GNU p
  - IDEs



mapping by section

- determine how “sections” separate
  - they separate?
  - lib
  - projects
  - low level
  - namespaces
  - ages
  - nventions
  - up
- s t
- n't obvious until
- worked with the project



mapping by dependency

- can you see architectural model?
  - layered, component, pipe/filter
- do dependencies match?
  - trace dependencies with tools
  - follow #includes, imports
  - call graphs
- quality of dependency
  - tied to quality of interface
  - cohesion / coupling
- maps effect propagation

mapping by control flow

- where is the entry point?
- where is the “main” hub of control?
  - linear, batch process
  - event loop
  - message queue
  - app framework, component interface
- is it threaded?
  - how well controlled are the threads?
  - is it actually thread safe?
  - is it clear what can & can’t be concurrent?
  - thread priorities

mapping by history

- the age of the code
  - when was it started?
  - when was it last modified?
  - mine revision control
- who wrote it
  - one author / many authors?
- do you have the latest version?
  - what branch are you working on?
  - do other branches have interesting (useful) stuff?
- the source it came from originally
  - download / vendor / other team
- where it is going?
  - internal, resubmit upstream, publish to licensees

select mapping tools

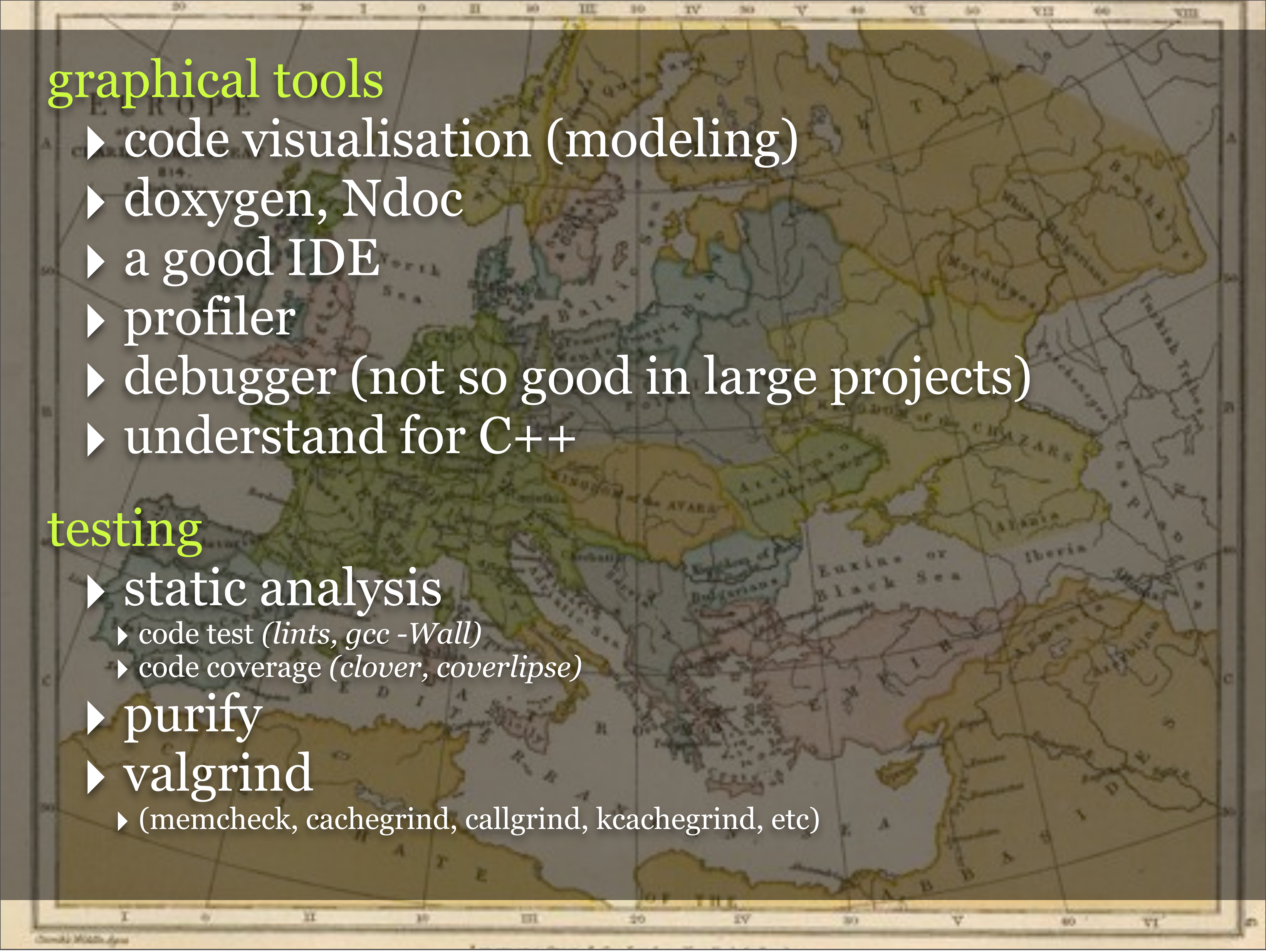
- command line
- graphical
- programatic

command line tools

- wc -l
- grep *(-i)*
- find *(-name)*
- xargs
- *piping*

```
ls -hF --color -R
find . -name “*.h” -o -name “*.c” | xargs cat | wc -l
find . -name “*.h” -o -name “*.c” | xargs grep -i “usb_debug”
```

- cygwin
- ctags *(excuberant ctags)*
- mlcscope

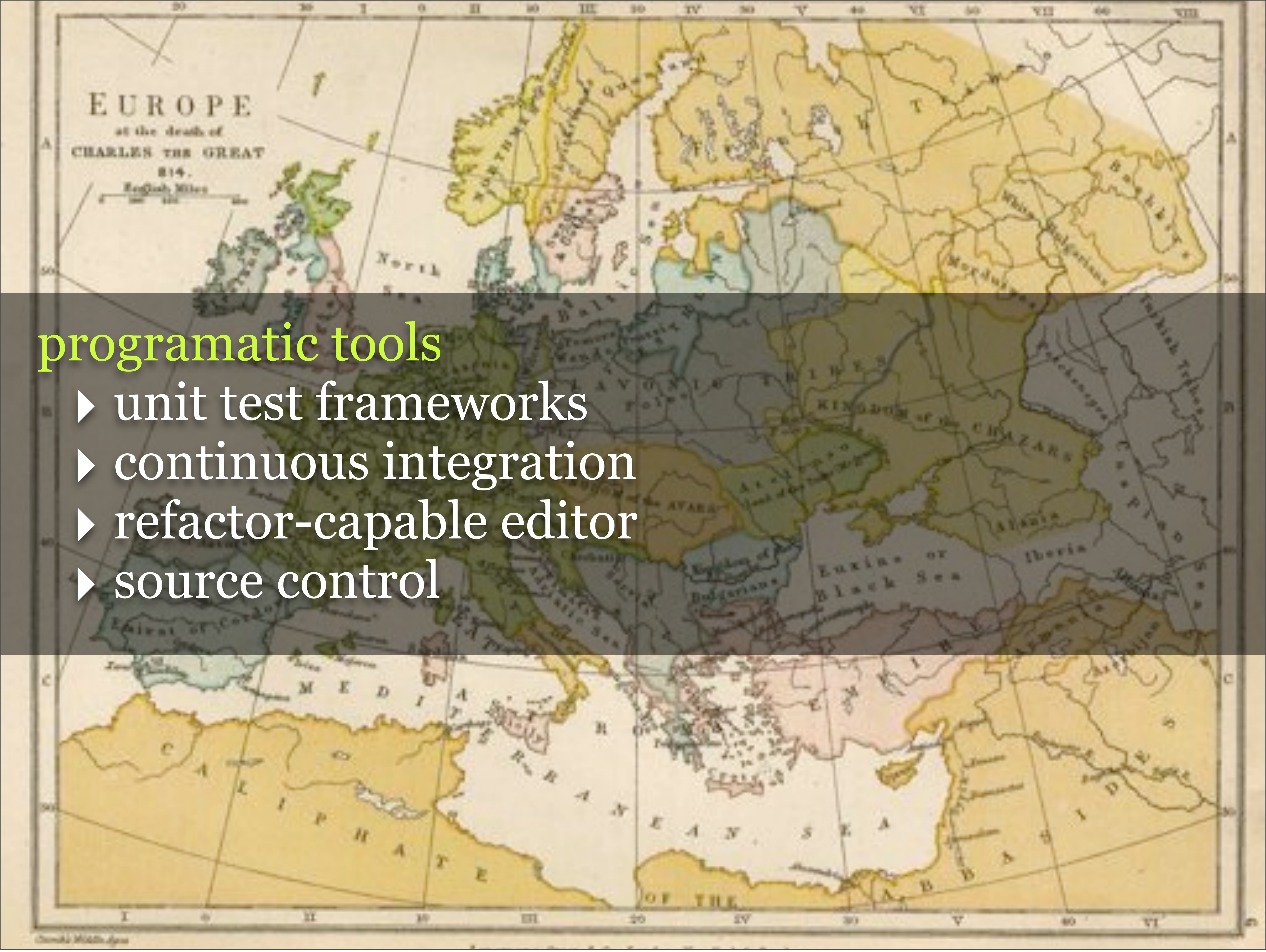


graphical tools

- code visualisation (modeling)
- doxygen, Ndoc
- a good IDE
- profiler
- debugger (not so good in large projects)
- understand for C++

testing

- static analysis
  - code test *(lints, gcc -Wall)*
  - code coverage *(clover, coverlipse)*
- purify
- valgrind
  - (memcheck, cachegrind, callgrind, kcachegrind, etc)



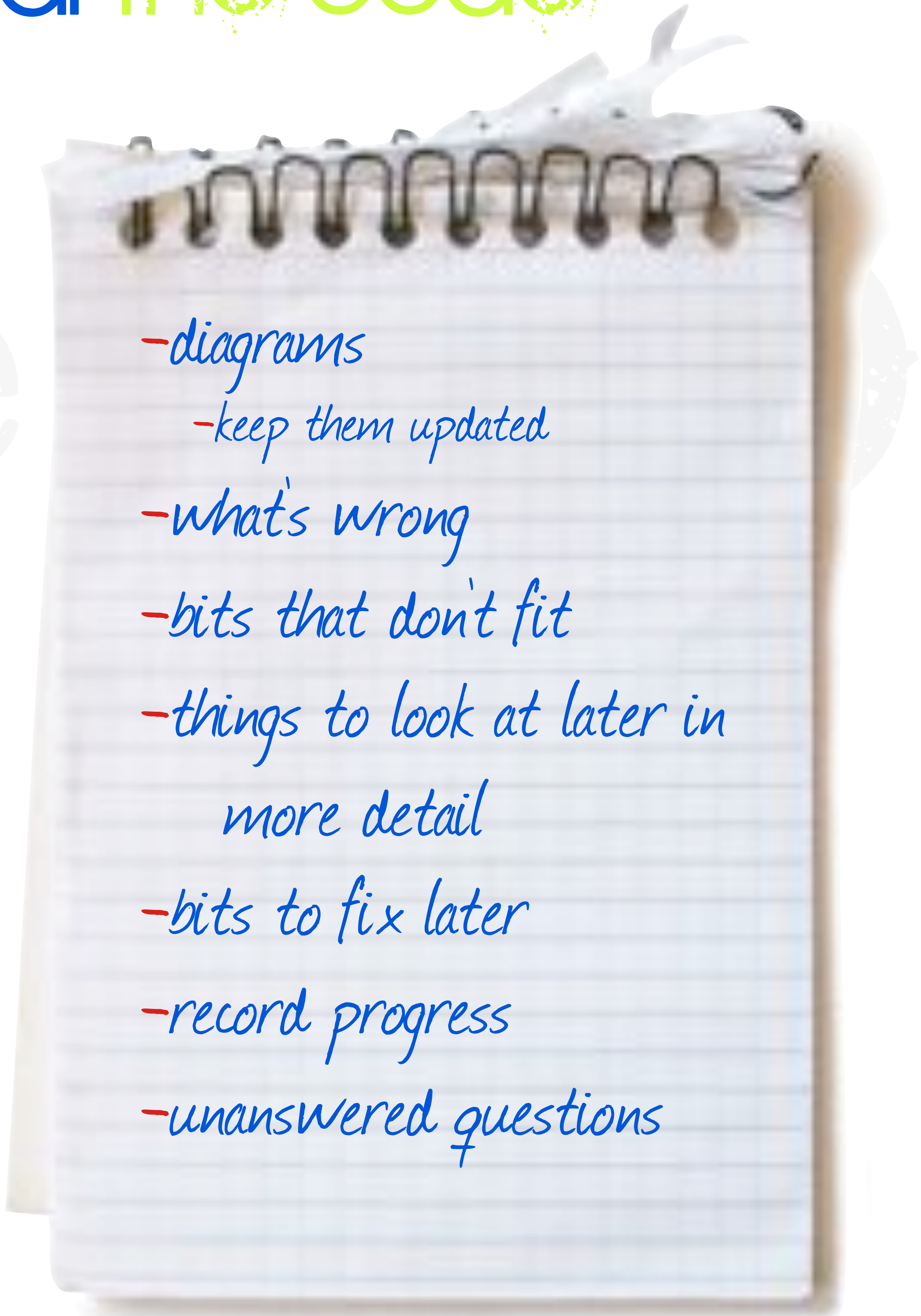
programatic tools

- unit test frameworks
- continuous integration
- refactor-capable editor
- source control

understand: the code

keep notes

- notebook
- wiki
- text files



- diagrams
  - keep them updated
- what's wrong
- bits that don't fit
- things to look at later in more detail
- bits to fix later
- record progress
- unanswered questions

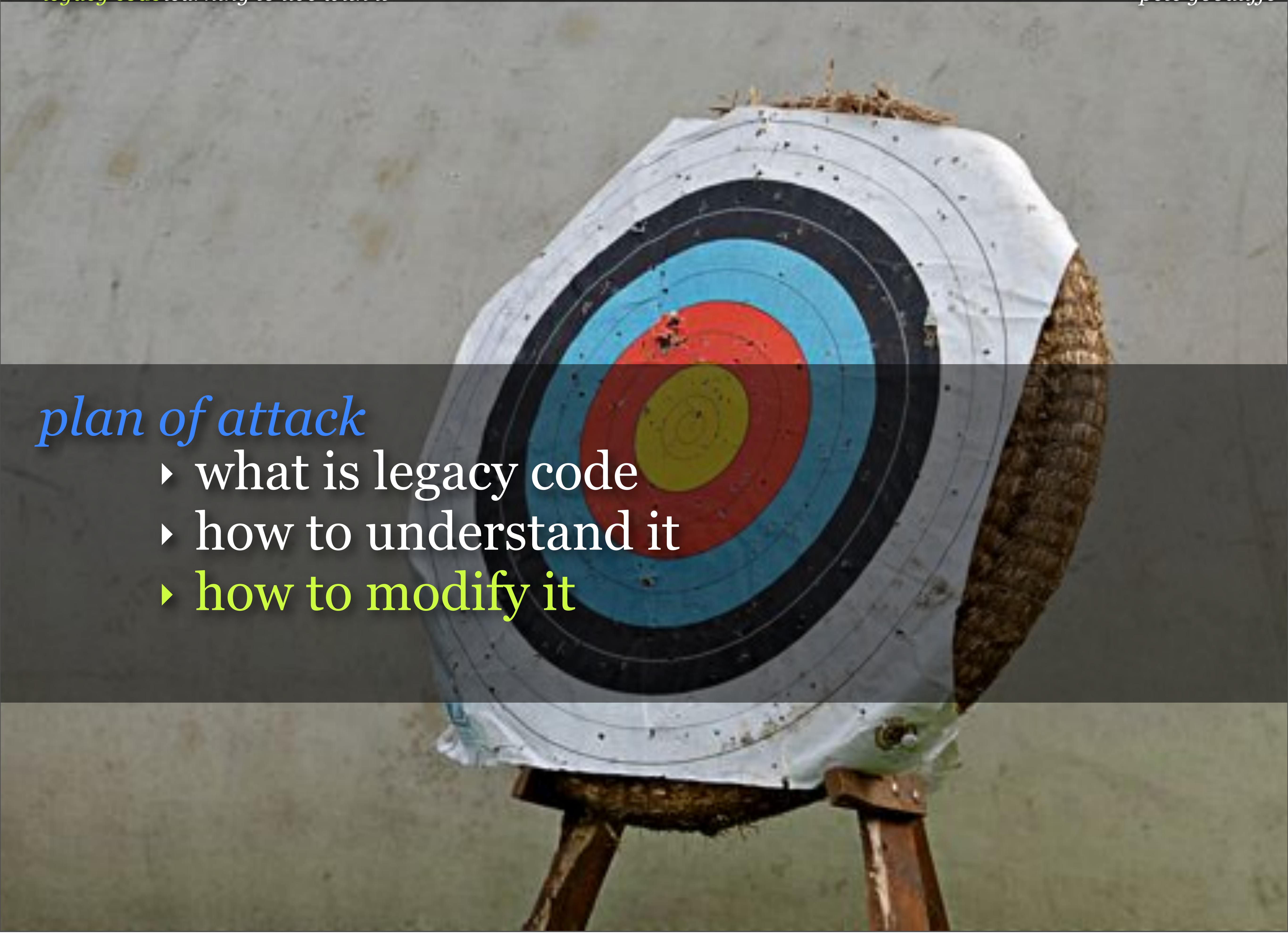
understand: the code

gauge quality

- structure
  - appropriateness
  - cohesion/coupling
  - single responsibility
- code quality
  - readability
  - for separate use
- the build
  - ease of building
  - documentation
  - automated (automatable)
  - does it build without warnings?

(the caution)

remember, this is not an *event*, its an ongoing *process*



plan of attack

- what is legacy code
- how to understand it
- how to modify it

this is the easy bit

well, not really

your mission

- make the changes
- don’t break anything
- improve the code on the way

your mission

- what are the requirements?
- *one task at a time*
- what else do you need to do?
- fix bugs
- refactor
- integrate
- one task at a time

- one task at a time

strategise

- Pinpoint the code to change
- Note locations for change
  - *Down to exact function(s)*
- What else might be affected by changes?
  - *Are you changing interfaces?*
- What kind of change is appropriate
  - *Wee fertle*
  - *Open heart surgery*
  - *Rip up and replace*
  - *Maintain old interface?*
- Experiment: try prototypes

write the code

write the code

but *that's* a different talk...

follow these rules >>

rule #1: code tact

- Follow the existing style
  - *Layout, naming, libraries*
    - *Add libraries carefully*
- Respect earlier programmers
  - *Whether still around or not*
- Treat the code carefully
  - *It’s a fragile beast*
  - *Be polite to it*
- Don’t ask too much of the code
  - *One thing at a time*

rule #2: know who to trust

- Don’t trust the build system
  - Rebuild, make clean, dependencies
  - *Especially if has custom steps*
- Not the earlier programmers
  - *Keep the benefit of the doubt*
- Not the specifications
  - *Documents get outdated*
- Only the code
  - *What it does right now*
  - *Know how to ask it*

rule #3: close the feedback loop

- Build it. Run it. Test it. Repeat.

  How long does it take?

- Break it

  Just to prove you've changed it

- Do one thing at a time

  Then you know what made the change

- Construct a test environment

  Don't stab in the dark

rule #3: close the feedback loop

- Avoid switching out
- Speed up turnaround
- Helps you get into flow
- Enables experimentation
- Prevents errors
- Slow turnaround kills development
  - Encourages multiple simultaneous changes
  - Switching tasks between builds

rule #3: close the feedback loop

- Prove your changes work
  - Nothing was broken
  - New functionality works
  - You have done what was required
- How do you do this?
  - testing
  - Needs a good covering
    - unit tests
    - acceptance tests

(tests)

- *don't* need to create 100% test coverage!
- more tests better than fewer
- broad coverage for main parts of functionality
  - *a few broad tests probably more effective than many narrow ones*
- targeted tests for the piece you're changing
- test-first for new code
- adding tests is not easy
  - *break out mockable interfaces*
  - *find/create seams to inspect behaviour*
  - *refactor*
  - *easier for OO code than procedural*
- explore existing functionality
  - *capture them in tests!*

rule #4: meddle methodically

- Tidy the house
  - *Don't leave commented out code*
  - *Delete unnecessary/old code*
  - *Comment clearly*
  - *Leave it as you'd like to live in*
- Minimise intrusion
  - *Only change what is necessary*
  - *Break out interfaces for change*
    - *Wrap and extend*
    - *Sprout functionality*
- Laziness: lean on the compiler
  - *Let the compiler help you make changes*

rule #4: meddle methodically

```
class Frog
{
    void Paint(Colour c);
    Food FavouriteFood() const;
};
```

- Tidy the house
  - Don't leave commented out code
  - Delete unnecessary/old code
  - Comment clearly
  - Leave it as you'd like to live in
- Minimise intrusion
  - Only change what is necessary
  - Break out interfaces for change
    - Wrap and extend
    - Sprout functionality
- Laziness: lean on the compiler
  - Let the compiler help you make changes

rule #4: meddle methodically

- Tidy the house
  - Don't leave commented out code
  - Delete unnecessary/old code
  - Comment clearly
  - Leave it as you'd like to live in
- Minimise intrusion
  - Only change what is necessary
  - Break out interfaces for change
    - Wrap and extend
    - Sprout functionality
- Laziness: lean on the compiler
  - Let the compiler help you make changes

```
class Frog
{
    void Paint(Colour c);
    Food FavouriteFood() const;
    static Colour GetLivery(Food f);
};
```

rule #4: meddle methodically

- Tidy the house
  - Don't leave commented out code
  - Delete unnecessary/old code
  - Comment clearly
  - Leave it as you'd like to live in
- Minimise intrusion
  - Only change what is necessary
  - Break out interfaces for change
  - Wrap and extend
  - Sprout functionality
- Laziness: lean on the compiler
  - Let the compiler help you make changes

```
class Frog
{
    void Paint(Colour c);
    Food FavouriteFood() const;
    static Colour GetLivery(Food f);
};

Colour c = Frog::GetLivery(freddie.FavouriteFood());
freddie.Paint(c);
```

rule #4: meddle methodically

- Tidy the house
  - Don’t leave commented out code
  - Delete unnecessary/old code
  - Comment clearly
  - Leave it as you’d like to live in
- Minimise intrusion
  - Only change what is necessary
  - Break out interfaces for change
    - Wrap and extend
    - Sprout functionality
- Laziness: lean on the compiler
  - Let the compiler help you make changes

```
class Frog
{
    void Paint(Colour c);
    Food FavouriteFood() const;
    static Colour GetLivery(Food f);
};

Colour c = Frog::GetLivery(freddie.FavouriteFood());
freddie.Paint(c);
```

rule #4: meddle methodically

- Tidy the house
  - Don't leave commented out code
  - Delete unnecessary/old code
  - Comment clearly
  - Leave it as you'd like to live in
- Minimise intrusion
  - Only change what is necessary
  - Break out interfaces for change
  - Wrap and extend
  - Sprout functionality
- Laziness: lean on the compiler
  - Let the compiler help you make changes

```
class Frog
{
    void XXX_Paint(Colour c);
    void PaintInLivery();
    Food FavouriteFood() const;
    static Colour GetLivery(Food f);
};

Colour c = Frog::GetLivery(freddie.FavouriteFood());
freddie.Paint(c); //< fails to compile
```

rule #4: meddle methodically

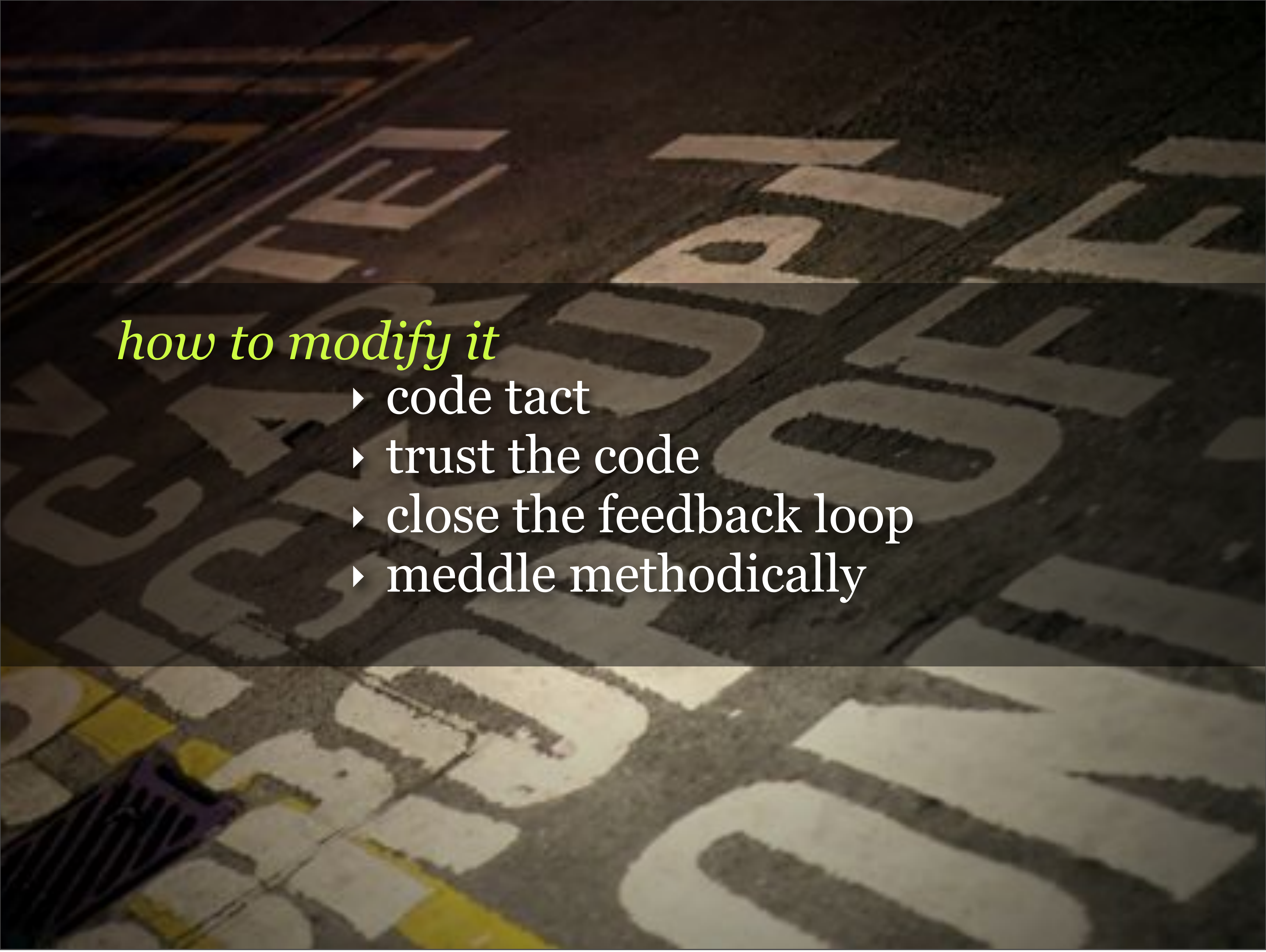
- Tidy the house
  - Don't leave commented out code
  - Delete unnecessary/old code
  - Comment clearly
  - Leave it as you'd like to live in
- Minimise intrusion
  - Only change what is necessary
  - Break out interfaces for change
    - Wrap and extend
    - Sprout functionality
- Laziness: lean on the compiler
  - Let the compiler help you make changes

```
class Frog
{
    void XXX_Paint(Colour c);
    void PaintInLivery();
    Food FavouriteFood() const;
    static Colour GetLivery(Food f);
};

freddie.PaintInLivery();
```

rule #4: meddle methodically

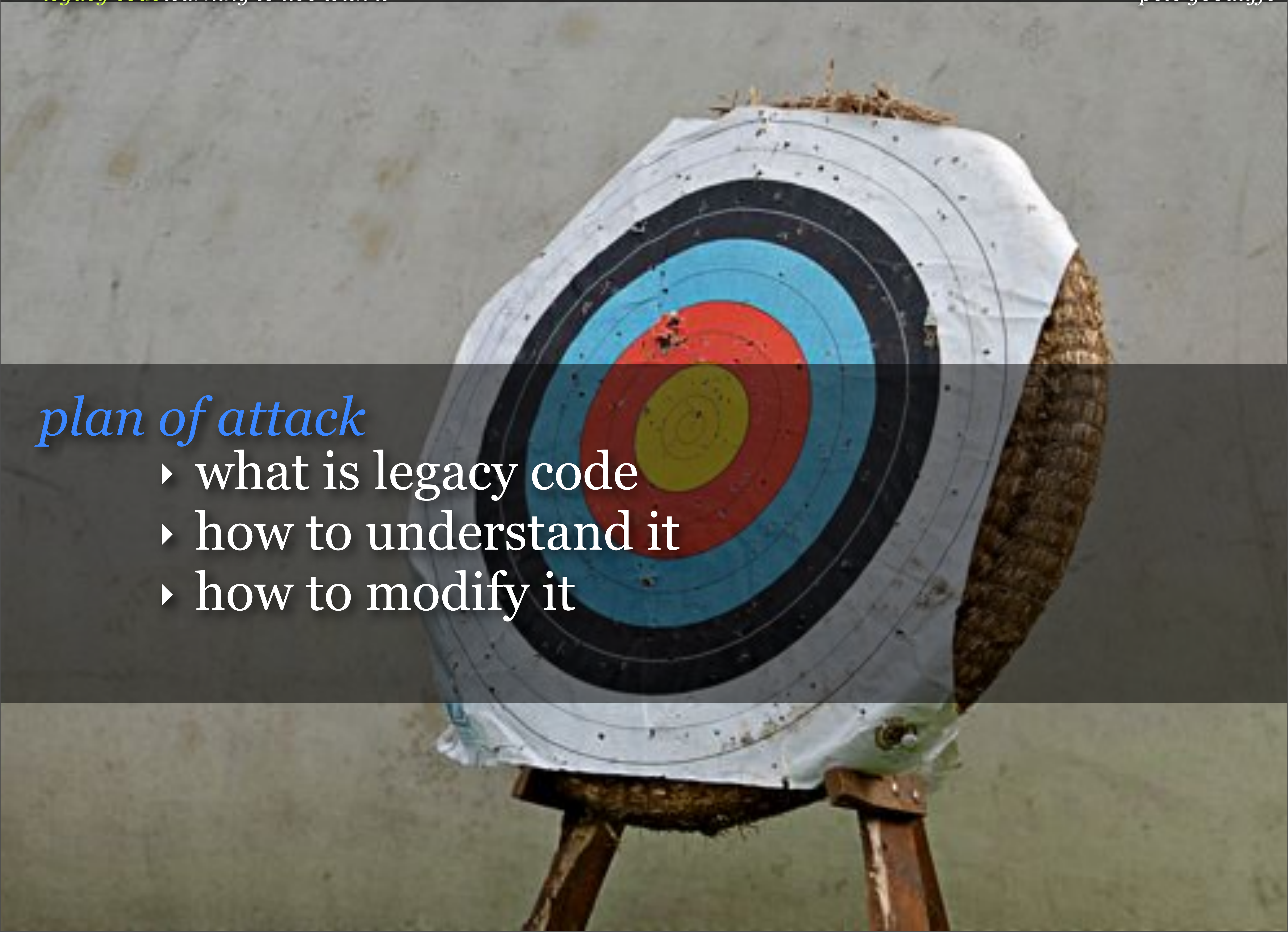
```
class Frog
{
    void Paint(Colour c);
    void PaintInLivery();
    Food FavouriteFood() const;
    static Colour GetLivery(Food f);
};

freddie.PaintInLivery();
```

- Tidy the house
  - Don't leave commented out code
  - Delete unnecessary/old code
  - Comment clearly
  - Leave it as you'd like to live in
- Minimise intrusion
  - Only change what is necessary
  - Break out interfaces for change
    - Wrap and extend
    - Sprout functionality
- Laziness: lean on the compiler
  - Let the compiler help you make changes



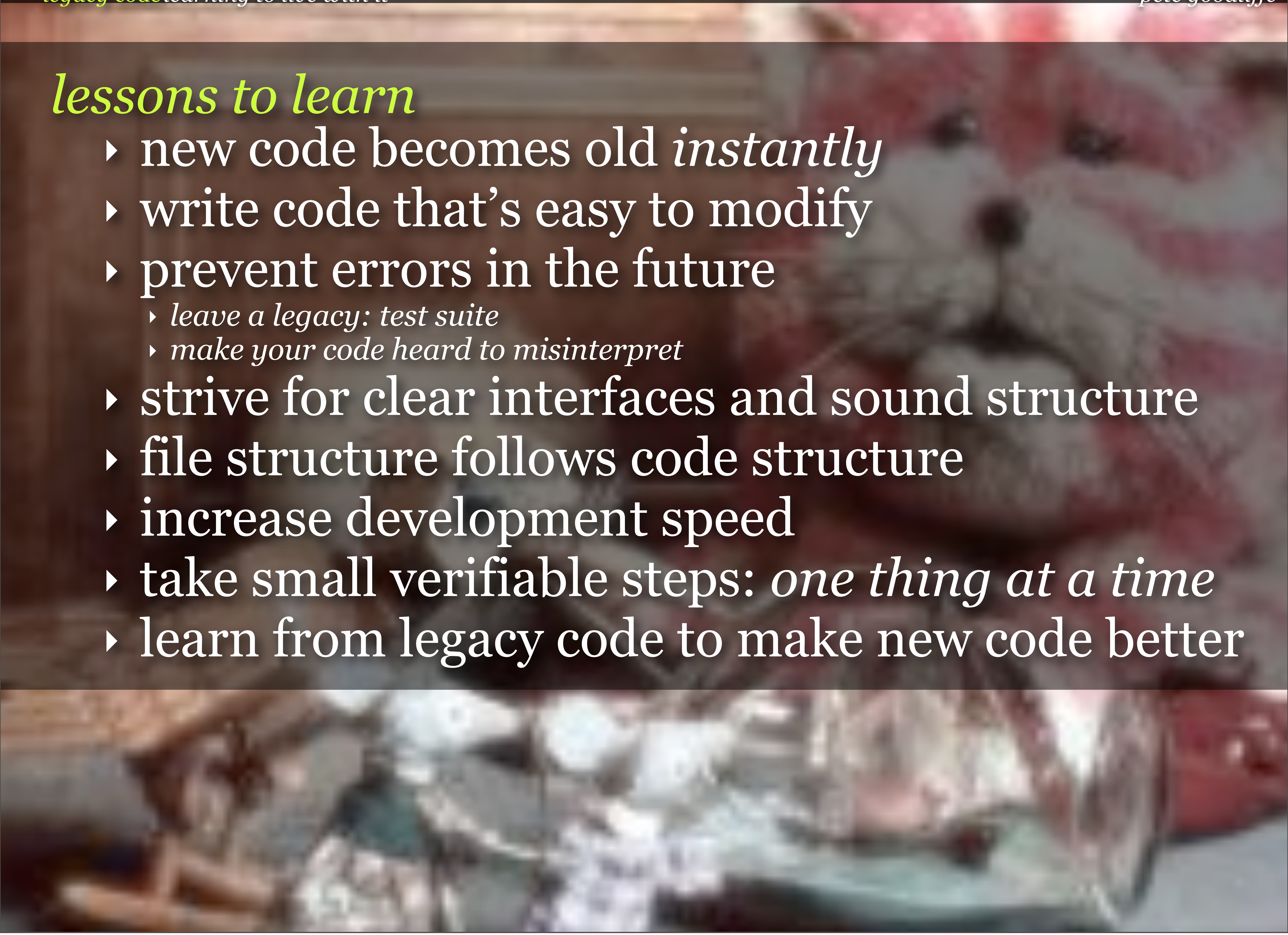
how to modify it

- code tact
- trust the code
- close the feedback loop
- meddle methodically



plan of attack

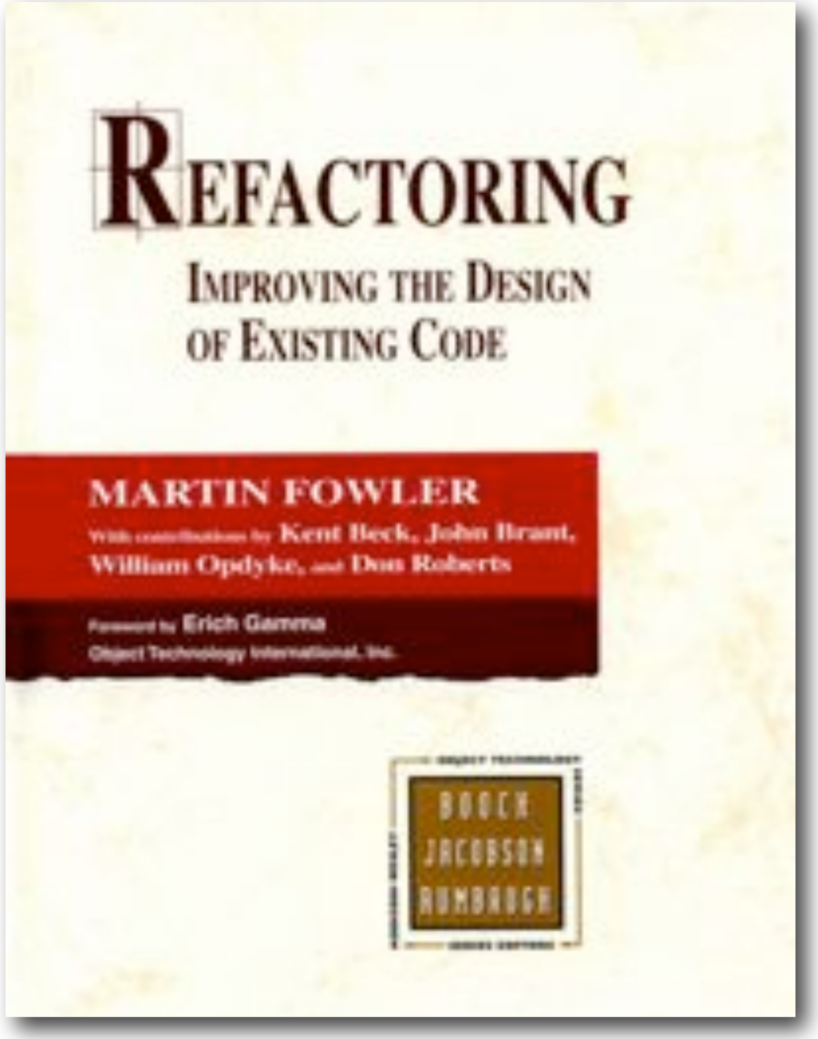
- what is legacy code
- how to understand it
- how to modify it

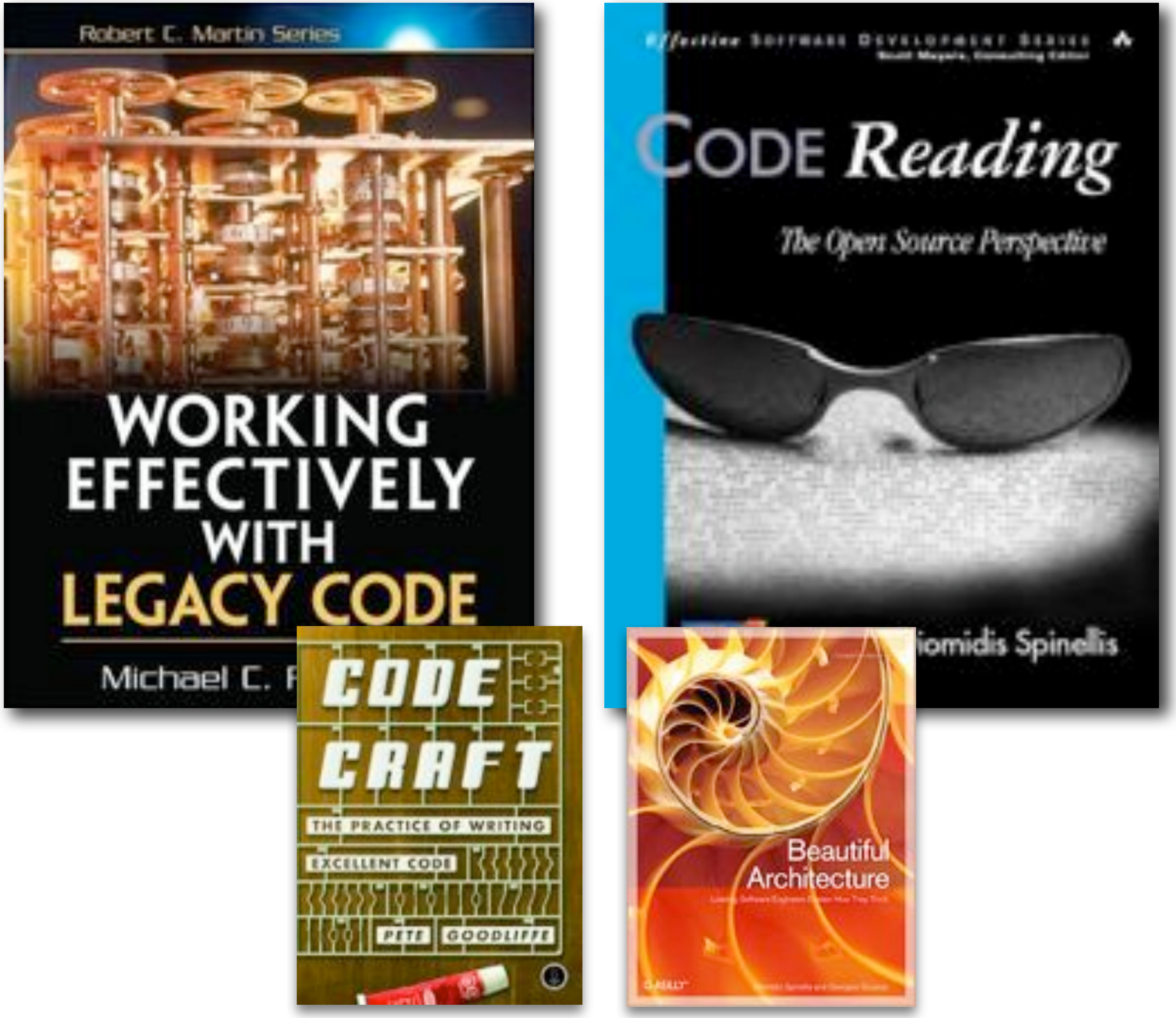


lessons to learn

- new code becomes old *instantly*
- write code that's easy to modify
- prevent errors in the future
  - *leave a legacy: test suite*
  - *make your code heard to misinterpret*
- strive for clear interfaces and sound structure
- file structure follows code structure
- increase development speed
- take small verifiable steps: *one thing at a time*
- learn from legacy code to make new code better

further reading


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International, Inc.
BOOCH JACOBSON RUMBAUGH


Robert C. Martin Series
WORKING EFFECTIVELY WITH LEGACY CODE
Michael C. F
Effective Software Development Series
Scott Meyers, Consulting Editor
CODE Reading
The Open Source Perspective
iomidis Spinellis
CODE CRAFT
THE PRACTICE OF WRITING EXCELLENT CODE
PETE GOODLIFFE
Beautiful Architecture
O'REILLY

the end

Pete Goodliffe
pete@goodliffe.net