accu 2022

ABSTRACTION PATTERNS:

MAKING CODE RELIABLY BETTER WITHOUT DEEP UNDERSTANDING

KATE GREGORY

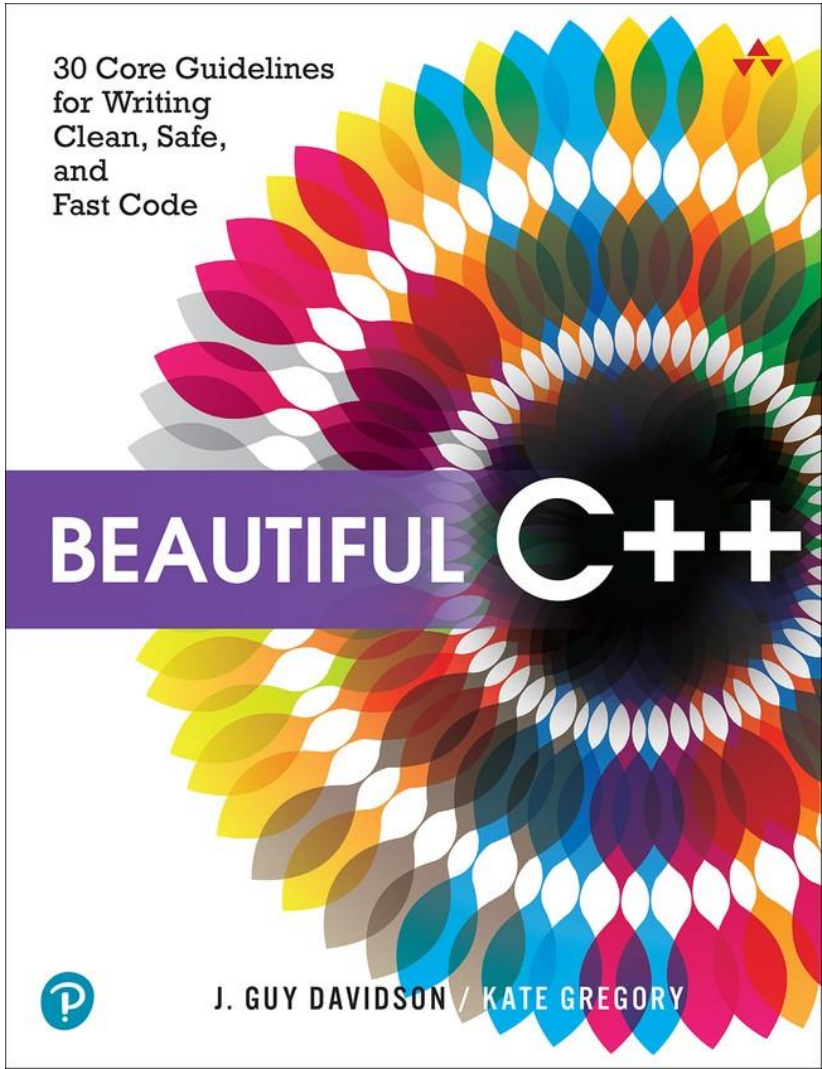
ABSTRACTION PATTERNS

Kate Gregory

kate@gregcons.com
www.gregcons.com/kateblog
@gregcons

FIRST, THANKS

- To Conor Hoekstra, for the truth about speaking
- To Guy Davidson, for Beautiful C++
- To Tony Van Eerd, for a SOLID talk at C++ Now 2021



WE TEACH ABSTRACTION LIES

- Abstractions are best discovered before any code is written
- It's important to learn notations for writing down abstractions (design)
- To create an abstraction, you need a deep understanding of the domain
  - business, engineering, science, regulations
- Once you've abstracted a system, you're all set

THE TRUTH

- Some abstractions are discovered during design
  - They do need a pre-code notation
  - It does require domain knowledge to discover them
- Finding those abstractions doesn't mean you're done
- You are likely to continue to find abstractions for the life of the software
- Finding those abstractions is done completely differently from the ones you find in design

COMPLETELY DIFFERENTLY

- You do not need domain knowledge
- You do not record these abstractions in some non-code notation
- You do not check with “the business” to see if you have them right

- They arise from the code
- You record them in the code

STORIES OF MY WORK

- I teach people Modern C++
- I work to reduce fear of C++
- I rescue failing projects
- I help teams who own code they can't understand or maintain

WHAT IS AN ABSTRACTION?

- Something with a name
  - Not just a class
- A way to reduce repetition and duplication
- A way to isolate parts of a problem
- Sometimes, something another person already wrote and tested
  - A lookup table, string, collection, command-line-option-parser
  - std::find(), std::all_of(), swap() … and of course, that's a rotate!
- Abstraction localizes and minimizes complexity

WHY PROVIDE MISSING ABSTRACTIONS?

Several thousand lines of code working with 50 member variables is not very far from global mutable state

EXAMPLES

LITERAL (MAGIC) NUMBERS BECOME NAMED CONSTANTS

- Local or in a class
- Often the name is in a comment nearby
- 0 or 1 exempted
- Make sure someone else hasn't done it for you
  - DaysPerWeek implies you should be converting durations with help from <chrono>
  - std::numbers::pi awaits you in <numbers>

MANY #DEFINES BECOME AN ENUM

```
#define RTB 1000 //basic report
#define RTC 2000 // customer-requested report
#define RTBC 2001 // customer-requested, for a big customer
```

- So many mistakes to make here

```
enum ReportTypes
{
	Basic,
	Customer,
	BigCustomer,
	// . . .
};
```

MANY PARAMS BECOME A STRUCT

```
Update(true, false, false, false, false, false, true);
// . .
Update(true, false, true, false, false, false, false);
```

- Consecutive parameters of the same type are a nightmare
- But too many, no matter the type, are unpleasant
- The more so if there is a usual or default value

```
struct UpdateOptions
{
	bool Europe = false;
	bool YearEnd = false;
	bool NoticesSent = false;
	bool IncludeLoans = false;
	bool RefreshTables = false;
	bool ExtraAuditEntries = false;
	bool Preliminary = false;
};

bool Update(UpdateOptions opt);
```

```
UpdateOptions EuropePrelim;
EuropePrelim.Europe = true;
EuropePrelim.Preliminary = true;
Update(EuropePrelim);
// . .
UpdateOptions EuropeSent;
EuropeSent.Europe = true;
EuropeSent.NoticesSent = true;
Update(EuropeSent);
```

OBVIOUS (BUT WRONG) GUESSES

```
void DrawRect(int, int, int, int);
```

- X and y co-ords? One xy plus a height and a width?

```
void DrawRect(Point, Point);
```

IS IT JUST A STRUCT?

- Passing 3 strings and a float becomes passing an abstraction
- Perhaps this free function that takes a T should actually be a member function of T

- Don't automatically add getters, setters etc
- Is there an invariant?
- Are there other nearby variables and functions that belong here?

OUT PARAMS ARE REPLACED WITH RETURNING A STRUCT

- Or possibly a pair, or a tuple
- Or std::optional or expected

VARIABLES WITH SIMILAR NAMES BECOME MEMBERS OF A CLASS

```
string empfirstname;
string emplastname;
string emptitle;
int empsalary;
```

- May actually be spread out a little, declared and initialized at once, etc
- Common prefix is typical "in the wild"

```
class Employee { string firstname; string lastname; string title; int
salary; /* ... */ };
```

CLUMPS OF VARIABLES BECOME OBJECTS

- Easy to spot when stuff is declared before initializing

```
int startval, endval;
int numpoints;
double avg;
double tolerance;

int lat, lon;
int alt;
bool visible;
bool secure;
```

100 LINES OF CODE BECOMES A FUNCTION

- Blank lines are again the key
- Comments are also a free clue
- Use your refactoring tool
  - Minimize parameters to and from the new function

- If you're left with a function that contains just 50 other function calls, consider another layer of abstraction
  - 5 functions that each call 10
  - Does that add value? Isolate the problem? Give things names?
  - Or just add confusion?

SIMILAR CLASSES USE INHERITANCE

- With or without polymorphism
- Put the commonality in the base class

SIMILAR FUNCTIONS BECOME A TEMPLATE

- Probably not your first reflex
  - Can I just pass some common base class of the things this function works with and use polymorphism?
  - Can I find something they all have (eg all containers have iterators) and pass that instead?
  - Can I put the commonality in a helper function and have each overload call it?
- These choices aren't wrong
- But a template is often cleaner

A CLASS WITH A "TYPE" MEMBER SWITCHES TO INHERITANCE

- There's nothing inherently wrong with having a "report type" member in the Report class, or "account type" member in the Account class
- But as a system grows, it can be a pain point
- Giant switch statements get longer as more types are added
- Some functions have no common lines, just the switch

```
void printHeader(Report const& r)
{
	// ... common stuff
	switch (r.getReportType())
	{
	case Basic:
		//basic header
		break;
	case Customer:
		//customer header
		break;
	case BigCustomer:
		//big customer header
		break;
	default:
		//nothing
		break;
	}
	// ... lots more common stuff
}
```

```
bool FooterNeeded(Report const& r)
{
	switch (r.getReportType())
	{
	case Basic:
		return true;
	case Customer:
	case BigCustomer:
		return false;
	default:
		assert(false); //you forgot to add a case above
	}
}
```

INHERITANCE

- Base class Report
- Derived: BasicReport, CustomerReport, …
- Small overrides:

```
bool BigCustomerReport::FooterNeeded()
{
    return false;
}
```

- Easy to write and read
- Make the functions pure virtual in the base: can't forget to do one
  - Or have a default implementation in the base class if you prefer

SPLITTING CLASSES

- If a class is hard to name, it's probably doing too much
  - Holds two or more smaller abstractions
  - Consider splitting it
- Looking at its responsibilities and what it tracks may be revealing

LOOK FOR WHITESPACE

- With or without comments
- Gaps in lists of private member variables
- Gaps in lists of public member functions
- Gaps between blocks of code

- These are arranging the items into groups. Listen to that

WHY NOT HAVE EVERYTHING TOGETHER?

- Because it makes things easy
- You start using things without thinking about what else is affected
- Perhaps everything else is affected
  - Consider global mutable state
- Separating things generally improves the design
  - Keep like with like
  - Explicitly pass information across boundaries
  - Minimize what goes across boundaries

SPLITTING A CLASS: WHERE DO I PUT THE FUNCTIONS?

- If a function works with 3 things from one clump and 4 from another, which new class should it be a member function of?
  - Imagine and consider both
  - Also consider a free function (or member function of the containing class) that takes an instance of each of the new classes
- Make the information connections explicit and obvious

HUGE INTERFACE

```
int getx();
int gety();
Color getforeground();
Color getbackground();
std::string gettext();
// . . .
```

SEGREGATED INTERFACE

```
Location getlocation();
Appearance getappearance();
Content getcontent();
```

- A Location has getx() and gety()
- An Appearance has getforeground() or getforegroundcolor()
- A Content has gettext()
- etc

THINK ABOUT CHANGES

- When the code that used getx and gety now needs another parameter…
  - Do you change Location?
  - Or does it need something that isn't Location?
  - Having to think about this means better design
- Programmers in a hurry just make things work
- Often take dependencies, or make changes that ripple
- Encapsulation and abstraction protect from that

SOME MORE PATTERNS AND THINGS TO LOOK FOR

MOVE WORK FROM THE PREPROCESSOR TO THE BUILD SYSTEM

```
#if defined WIN32
    auto a_pressed = bool{GetKeyState('A') & 0x8000 != 0};
#elif defined LINUX
    auto a_pressed = /*really quite a lot of code*/
#endif
```

- Becomes

```
#include "keypress.h"
//…
auto a_pressed = key_state('A');
```

- And your **build process** includes and links the appropriate library for each platform you target

LOOK FOR THE WORD "AND"

- GetWidthAndHeight
  - You're missing an abstraction
  - GetSize
- FindLimitsAndAverage (to be "efficient")
  - Still missing an abstraction
  - UpdateCachedValues

USE THE ABSTRACTIONS YOU HAVE

BE AS SPECIFIC AS POSSIBLE

```
bool any = false;
for (unsigned int i = 0; i < nums.size(); ++i)
{
	if (nums[i] == 3)
	{
		any = true;
		break;
	}
}
```

BE AS SPECIFIC AS POSSIBLE

```
bool any = false;
for (auto n:nums)
{
      if (n == 3)
      {
            any = true;
            break;
      }
}
```

BE AS SPECIFIC AS POSSIBLE

```
auto it = std::find(begin(nums), end(nums), 3);
bool any = (it != end(nums));
```

BE AS SPECIFIC AS POSSIBLE

```
bool any = std::any_of(begin(nums), end(nums),
                       [](auto n) {return n == 3; });
```

HOW IS THAT AN ABSTRACTION?

- The ranged for is an abstraction compared to a for loop. It holds the idea of doing something exactly once to each element of a collection
- std::find is an abstraction compared to a loop. It combines the loop with the == operator
- std::any_of is an abstraction compared to find. It includes the idea that at least one instance was found
- As we use more specific and precise abstractions, we gain information (in the names) and shed work (because some of the work is now inside the abstraction)

CALL TO ACTION

- Look at your code with new eyes
  - Not just your ancient legacy code, but what you write this week
- Are there abstractions hiding there?
- Could you make it more readable by providing them?
- Can you convert a comment to a name?
- Are gaps and whitespace screaming at you?
- What can you gain by spotting abstraction patterns?