IC Coder's Club

Design patterns (in C++)

Andrew W. Rose

Simple exercise

You have been tasked to move <u>this pile</u> → from A to B

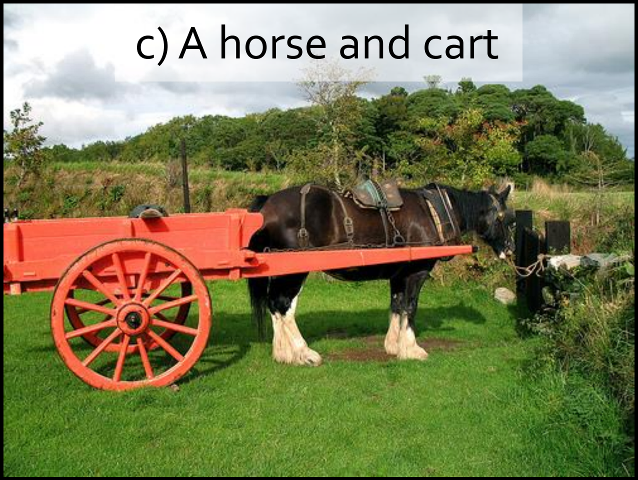
Simple exercise

You have been tasked to move this pile from A to B

You have three resources available to you:

a) Bare hands

b) A stack of timber



c) A horse and cart

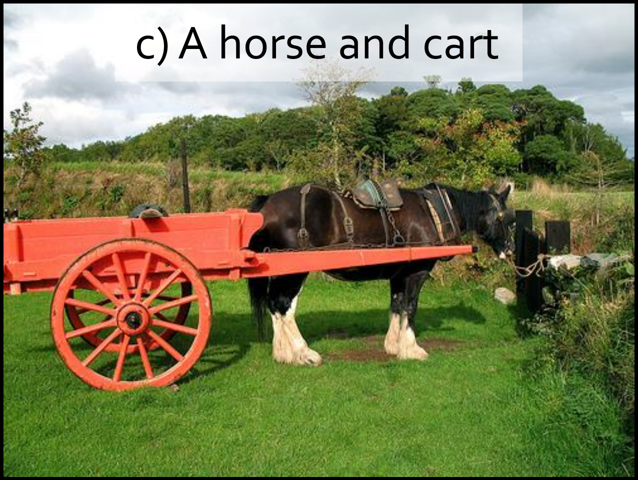
Simple exercise

You have been tasked to move this pile from A to B

You have three resources available to you:

a) Bare hands

b) A stack of timber



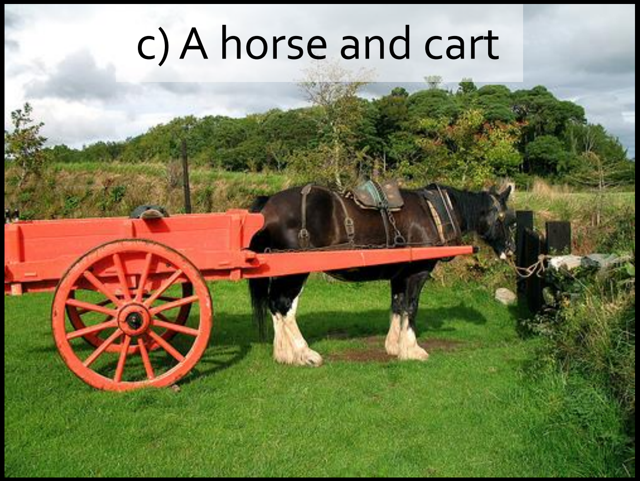
c) A horse and cart

How do you achieve the task in the quickest, least-painful way, which won't leave you up-to-your-neck in the produce you are moving, nor smelling of it?

Software analogy

a) Bare hands

b) A stack of timber

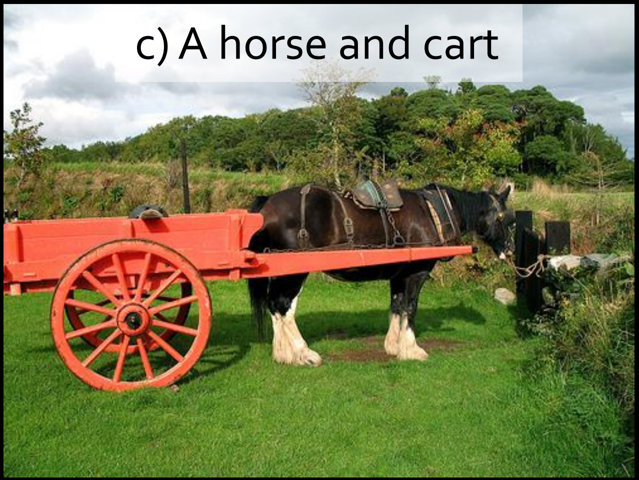


c) A horse and cart

Software analogy


a) Bare hands


b) A stack of timber

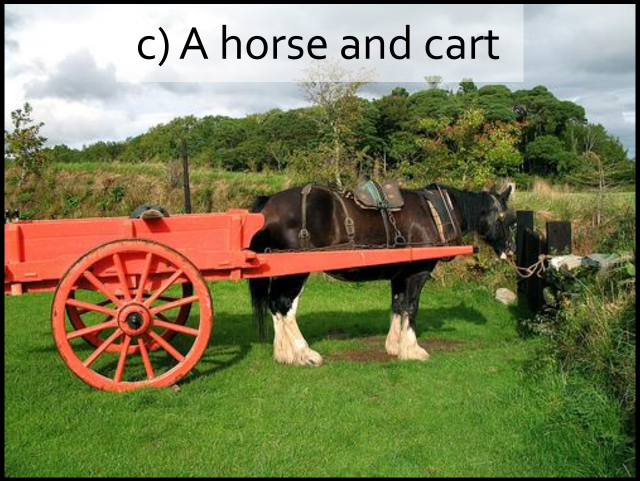

c) A horse and cart

Do a task manually

Software analogy


a) Bare hands


b) A stack of timber


c) A horse and cart

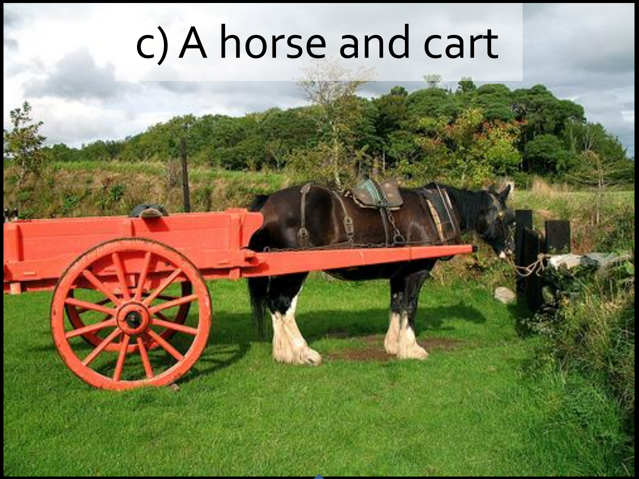
Do a task manually

Design the tools yourself

Software analogy


a) Bare hands


b) A stack of timber


c) A horse and cart

Do a task manually

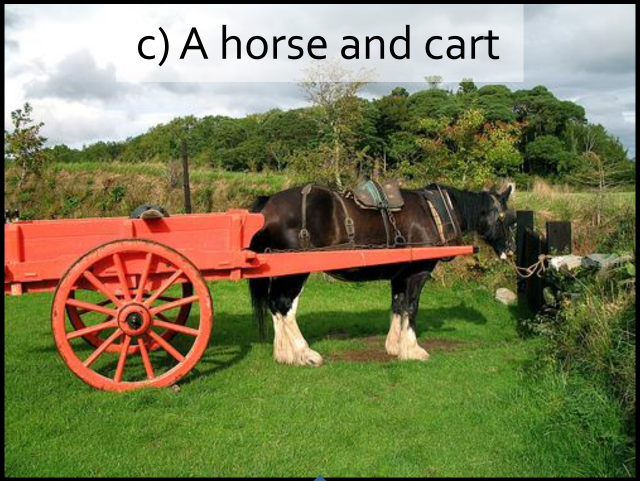
Design the tools yourself

Benefit from someone else's hard work

Software analogy


a) Bare hands


b) A stack of timber


c) A horse and cart

This is the purpose of design patterns!

Benefit from someone else's hard work

Motivation

I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important: Bad programmers worry about the code; good programmers worry about data structures and their relationships.

Linus Torvald

"Code and fix" development is not so much a deliberate strategy as an artefact of naïveté and schedule pressure on software developers.

Steve McConnell

Motivation

I will, in fact, claim that the difference between a bad programmer

a

struct

de;

vald

"Cod

as

an artefact of naïveté and schedule pressure on software developers.

Steve McConnell

Stopping and thinking before you write a single line of code will save you time, effort and inconvenience in future.

Software Design Patterns: What are they not?

- Magic
- The work of superhuman intelligence
- Necessary in all languages (some patterns are related to working around the constraints of the language itself)

Software Design Patterns: What are they?

- General reusable solutions to commonly occurring problem
- Formalized best practices
- A set of relationships and interactions between conceptual or example classes or objects, which say nothing about the final application classes or objects that the programmer will actually implement.
- Daunting at first
- A guaranteed way to increase the complexity of your code unnecessarily if you use them incorrectly or inappropriately

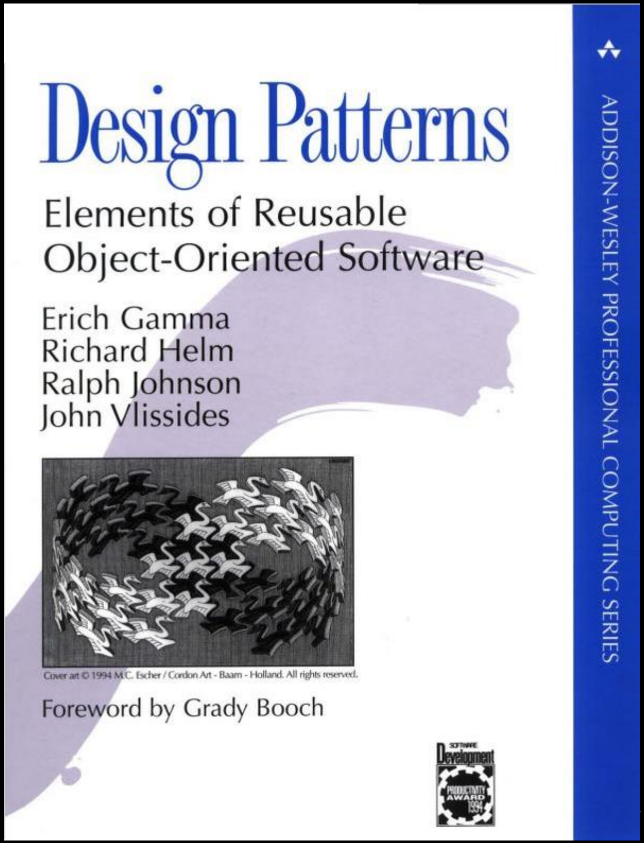
Software Design Patterns

I was told that the point of Coder's Club was to provide examples that couldn't be found in books

Software Design Patterns

I was told that the point of Coder’s Club was to provide examples that couldn’t be found in books

Ironic, given that the whole point of design patterns is that they are examples written in books



Design Patterns

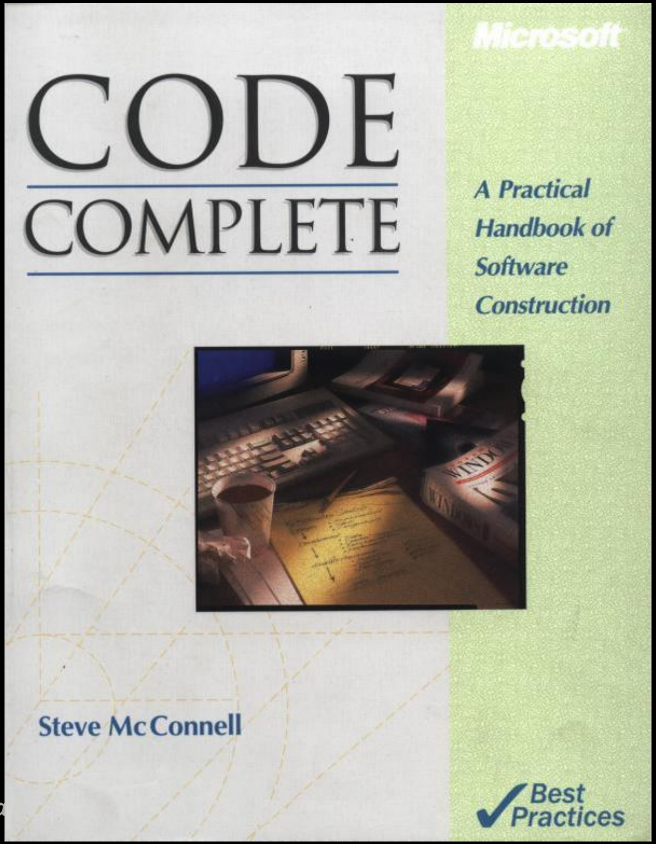
Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



Microsoft

CODE COMPLETE

A Practical Handbook of Software Construction

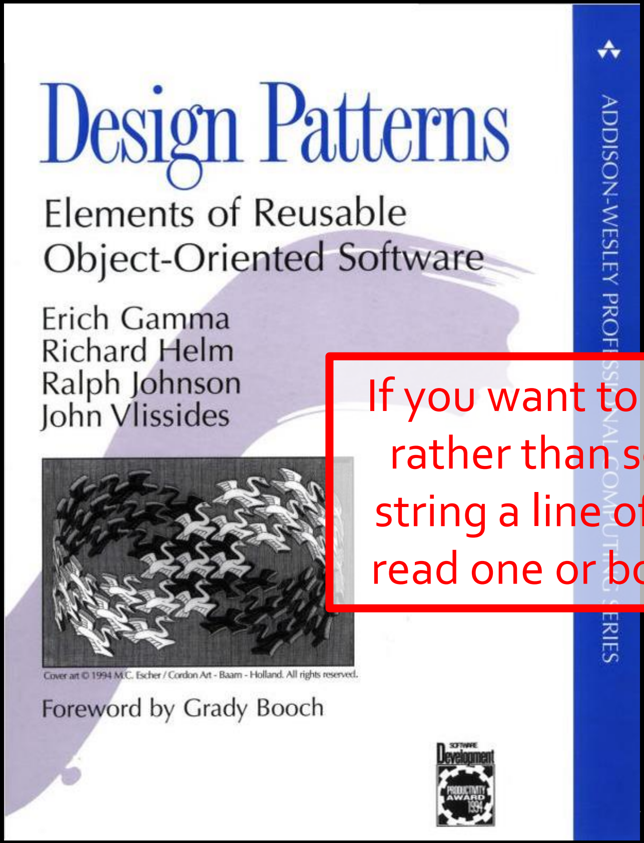
Steve McConnell

Best Practices

Software Design Patterns

I was told that the point of Coder's Club was to provide examples that couldn't be found in books

Ironic, given that the whole point of design patterns is that they are examples written in books



Design Patterns

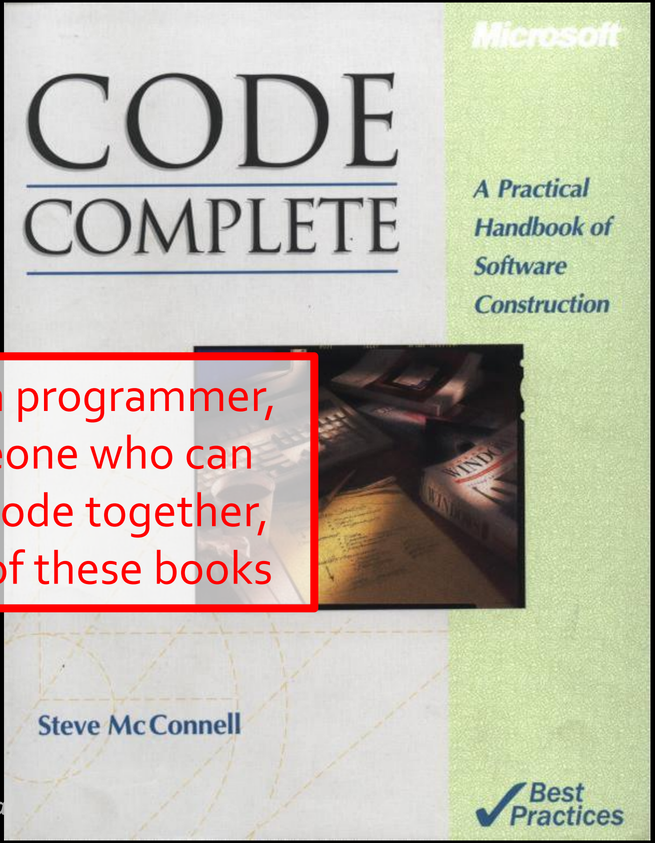
Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



Microsoft

CODE COMPLETE

A Practical Handbook of Software Construction

Steve McConnell

Best Practices

If you want to be a programmer, rather than someone who can string a line of C-code together, read one or both of these books

Software Design Patterns: Daunting

Abstract factory
Builder
Factory method
Lazy initialization
Multiton
Object pool
Prototype
Resource acquisition is initialization
Singleton
Adapter or Wrapper or Translator.
Bridge
Composite
Curiously recursive template pattern
Decorator
Facade
Flyweight
Front Controller
Module
Proxy
Twin

Blackboard
Chain of responsibility
Command
Interpreter
Iterator
Mediator
Memento
Null object
Observer or Publish/subscribe
Servant
Specification
State
Strategy
Template or Hollywood method
Visitor

⋮

Software Design Patterns

Creational

- Abstract factory
- Builder
- Factory method
- Lazy initialization
- Multiton
- Object pool
- Prototype
- Resource acquisition is initialization
- Singleton

Structural

- Adapter or Wrapper or Translator.
- Bridge
- Composite
- Curiously recursive template pattern
- Decorator
- Facade
- Flyweight
- Front Controller
- Module
- Proxy
- Twin

Behavioural

- Blackboard
- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Null object
- Observer or Publish/subscribe
- Servant
- Specification
- State
- Strategy
- Template or Hollywood method
- Visitor

⋮

Factory method, Builder and Abstract factory patterns (For when a constructor just won’t cut it)

A factory is a trivial concept – don’t call the object constructor directly, call a function which does it for you.

Three most common non-trivial examples are:

- Factory method
- Builder
- Abstract factory

Factory method, Builder and Abstract factory patterns (For when a constructor just won’t cut it)

Consider a set of classes which differ only by the concrete implementation of their member variables.

Because they are otherwise identical, it is appropriate for these classes to inherit from a base class.

The constructor of the class may be very complicated and nevertheless it would be wholly inappropriate to expect all the concrete implementations of the class to copy-paste-and-modify the constructor.

Factory method, Builder and Abstract factory patterns (For when a constructor just won't cut it)

Consider a set of classes which differ only by the concrete implementation of their member variables.

Because they are otherwise identical, it is appropriate for these classes to inherit from a base class.

The constructor of the class may be very complicated and nevertheless it would be wholly inappropriate to expect all the concrete implementations of the class to copy-paste-and-modify the constructor.

The factory method helps:

- The base class includes a pure virtual method for creating the member variables.
- The base class can do all the nastiness, safe in the knowledge that…
- All concrete implementations have to implement the factory method

Factory method, Builder and Abstract factory patterns (For when a constructor just won’t cut it)

```
class BaseClass {
public:
  BaseClass(){ …Nastiness…Complexity… makeObject() …More nastiness & complexity…Yuk…Yuk…Yuk… }
  virtual AbstractMemberType* makeObject() = 0;
  AbstractMemberType* mMember;
};

class ImplementationA : public BaseClass {
public:
  ImplementationA() : BaseClass() { …Simplicity… }
  virtual AbstractMemberType* makeObject() { return new MemberTypeA; }
};

class ImplementationB : public BaseClass {
public:
  ImplementationB() : BaseClass() { …Simplicity… }
  virtual AbstractMemberType* makeObject() { return new MemberTypeB; }
};
```

Factory method, Builder and Abstract factory patterns (For when a constructor just won’t cut it)

```
class BaseClass {
public:
  BaseClass(){ …Nastiness…Complexity… makeObject() …More nastiness & complexity…Yuk…Yuk…Yuk… }
  virtual AbstractMemberType* makeObject() = 0;
  AbstractMemberType* mMember;
};

class ImplementationA : public BaseClass {
public:
  ImplementationA() : BaseClass() { …Simplicity… }
  virtual AbstractMemberType* makeObject() { return new MemberTypeA; }
};

class ImplementationB : public BaseClass {
public:
  ImplementationB() : BaseClass() { …Simplicity… }
  virtual AbstractMemberType* makeObject() { return new MemberTypeB; }
};
```

See, no superhuman intelligence required here

Factory method, Builder and Abstract factory patterns (For when a constructor just won't cut it)

Often, designs start out using
Factory Method (less complicated, more customizable, subclasses proliferate)
and evolve toward
Abstract Factory, Prototype, or Builder (more flexible, more complex)
as the designer discovers where more flexibility is needed.
[Design Patterns pp. 92]

Factory method, Builder and Abstract factory patterns (For when a constructor just won’t cut it)

Consider a class which has a very large set of independent options which should be defined at construction time and then be immutable.

This could result in a very large number of permutations of constructors

Alternatively end up with a lot of “Set…()” methods in the class and depend on the honesty/intelligence of the end user not to use them (yeah, right)

Factory method, Builder and Abstract factory patterns (For when a constructor just won't cut it)

Consider a class which has a very large set of independent options which should be defined at construction time and then be immutable.

This could result in a very large number of permutations of constructors

Alternatively end up with a lot of "Set...()" methods in the class and depend on the honesty/intelligence of the end user not to use them (yeah, right)

A Builder is a friendly class with all the Set-option method and a single get method which returns the fully-formed object

Factory method, Builder and Abstract factory patterns (For when a constructor just won’t cut it)

```
class MultiOptionClass {
private:
 friend class MultiOptionClassBuilder;
 MultiOptionClass(){}
};

class MultiOptionClassBuilder {
public:
 MultiOptionClassBuilder() {}
 void SetOptionA(…){}
 void SetOptionB(…){}
       :
 void SetOptionN(…){}
 MultiOptionClass getMultiOptionClass() { ….Construct class and apply options… }
};
```

Factory method, Builder and Abstract factory patterns (For when a constructor just won't cut it)

Suppose you have a perfectly-formed abstract base class and associated concrete implementations.

Since the base class is abstract, we tend to know what type of object we have created, since we must chose a concrete implementations to instantiate.

In many cases, this kind of defeat the point of having an abstract base class…

An Abstract Factory helps out

Abstract factory case study: uHAL

uHAL is a library developed for LHC upgrades

It is a library which provides tools for describing the structure of registers within hardware and for configuring hardware either directly or indirectly over Gigabit Ethernet.

Abstract factory case study: uHAL

uHAL is a library developed for LHC upgrades

It is a library which provides tools for describing the structure of registers within hardware and for configuring hardware either directly or indirectly over Gigabit Ethernet.

All configurations are stored in XML files/databases

All the user wants to know is their board's name. The end user should not need to know how they are talking to their hardware, which protocol version they are using, etc. Their software should be agnostic to all that nonsense…

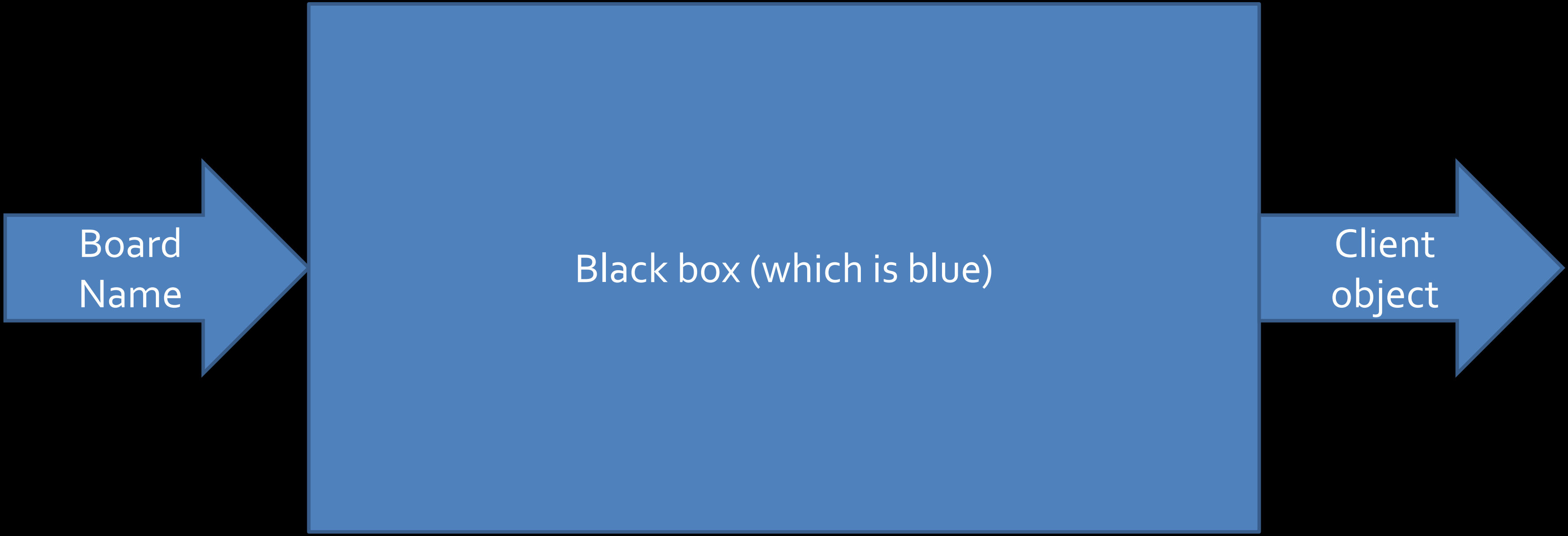
Sounds like an ideal candidate for an abstract base class…

Abstract factory case study: uHAL

9 protocol variants denoted by the protocol field within the URI:
yyy://xxx.xxx.xxx.xxx/…….

Each variant requires a different class to handle it

All the user wants to see is a (pointer to a) Client object (which, trust me, they never, ever, ever want to see inside)



Board Name

Black box (which is blue)

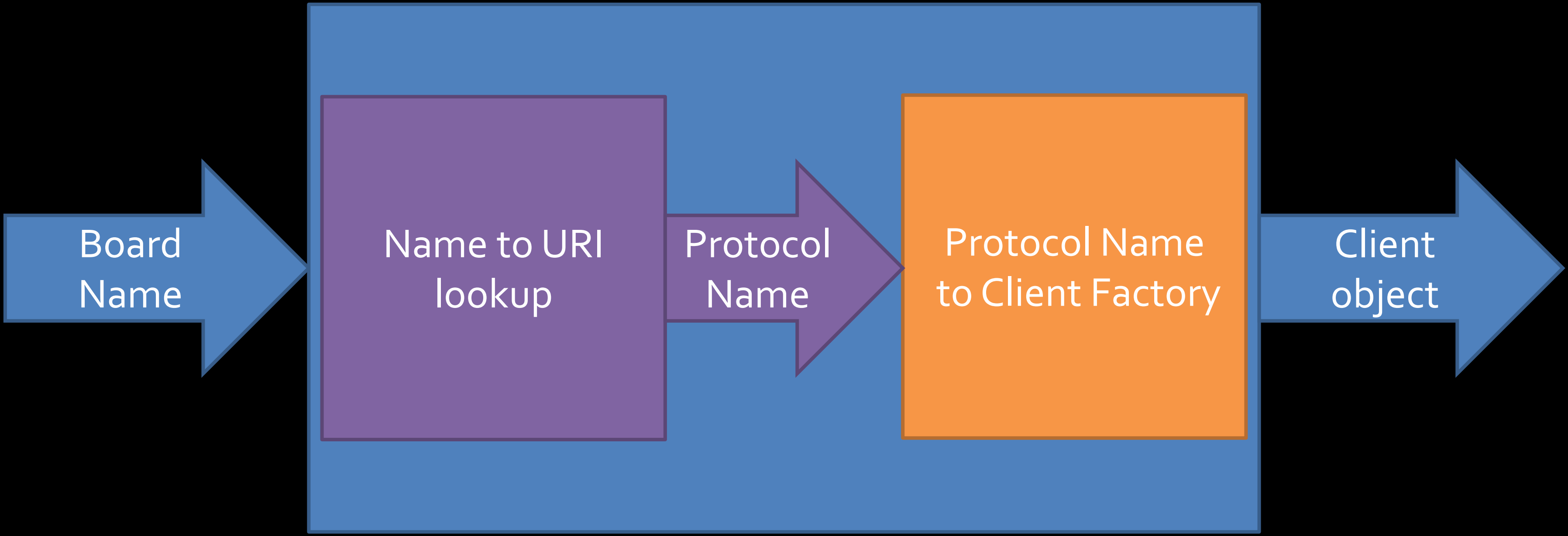
Client object

Abstract factory case study: uHAL

9 protocol variants denoted by the protocol field within the URI:
yyy://xxx.xxx.xxx.xxx/…….

Each variant requires a different class to handle it

All the user wants to see is a (pointer to a) Client object (which, trust me, they never, ever, ever want to see inside)



Board Name

Name to URI lookup

Protocol Name

Protocol Name to Client Factory

Client object

Abstract factory case study: uHAL

The problem: convert a string to a class type

Also: Keep the interface clean for adding more protocols later

```
class ClientFactory {
public:
 Client* create( const std::string& aProtocol );
 template <class Protocol> void addProtocol( const std::string& aProtocol );
 ClientFactory();
};
```

Abstract factory case study: uHAL

The problem: convert a string to a class type

Also: Keep the interface clean for adding more protocols later

```
class ClientFactory {
public:
 Client* create( const std::string& aProtocol );
 template <class Protocol> void addProtocol( const std::string& aProtocol );
 ClientFactory();
};
```

Adding protocols is as simple as

```
addProtocol< ProtocolA > ( “ProtocolA” );
addProtocol< ProtocolB > ( “ProtocolB” );
addProtocol< ProtocolC > ( “ProtocolC” );
```

So definitely meets the second criterion

Abstract factory case study: uHAL

To construct an object of a particular concrete type, the factory needs a worker who knows about that type

Use templates!

```
class FactoryWorkerInterface {
public:
 Client* create() = 0;
};

template <class Protocol>
class FactoryWorkerImplementation {
public:
 Client* create(){ return new Protocol; }
};
```

Abstract factory case study: uHAL

The factory can then associate a string with a worker object using a standard (hash) map:

```
std::map< std::string , FactoryWorkerInterface* > mListOfWorkers;
```

The ClientFactory create() function then simply passes the job to the appropriate worker:

```
Client* ClientFactory::create( const std::string& aProtocol ){
  return mListOfWorkers[ aProtocol ] -> create();
}
```

Neither the user nor, in fact, the factory ever see the pointer to the concrete object, only the pointer to the abstract Client.

The Singleton pattern

Let us consider the factory we have just created:

Is there ever a use case for having more than one copy of this factory?

The Singleton pattern

Let us consider the factory we have just created:

Is there ever a use case for having more than one copy of this factory? NO!

Is there a good reason not to have multiple copies of this factory?

The Singleton pattern

Let us consider the factory we have just created:

Is there ever a use case for having more than one copy of this factory? NO!

Is there a good reason not to have multiple copies of this factory? YES!

In our example the map only has 9 entries but it could, in principle, have many thousands of entries. We do not want to fill this map many times over.

The Singleton pattern

Let us consider the factory we have just created:

Is there ever a use case for having more than one copy of this factory? NO!

Is there a good reason not to have multiple copies of this factory? YES!

In our example the map only has 9 entries but it could, in principle, have many thousands of entries. We do not want to fill this map many times over.

One option is to create a global copy of the factory but global variables are evil

- They pollute the global namespace
- Consume resources even if not used
- Are inherently unsafe
- Do not stop the user creating a second copy of the factory anyway

The Singleton pattern

Let us consider the factory we have just created:

Is there ever a use case for having more than one copy of this factory? NO!

Is there a good reason not to have multiple copies of this factory? YES!

In our example the map only has 9 entries but it could, in principle, have many thousands of entries. We do not want to fill this map many times over.

One option is to create a global copy of the factory but global variables are evil

- They pollute the global namespace
- Consume resources even if not used
- Are inherently unsafe
- Do not stop the user creating a second copy of the factory anyway

Use the Singleton pattern

The Singleton pattern

```
class SingletonClass {
private:
  SingletonClass(){}
  static SingletonClass* mInstance;
public:
  static SingletonClass& getInstance()
  {
    if( !mInstance )
    {
       mInstance = new SingletonClass;
       … Initialize the Singleton Class …
    }
    return *mInstance;
  }
};

SingletonClass* SingletonClass::mInstance = NULL;
```

The Singleton pattern

```
class SingletonClass {
private:
  SingletonClass(){}
  static SingletonClass* mInstance;
public:
  static SingletonClass& getInstance()
  {
    if( !mInstance )
    {
      mInstance = new SingletonClass;
      … Initialize the Singleton Class …
    }
    return *mInstance;
  }
};

SingletonClass* SingletonClass::mInstance = NULL;
```

The constructor is private
The class contains a static pointer to itself

Remembering to instantiate
the static member variable

The Singleton pattern

```
class SingletonClass {
private:
  SingletonClass(){}
  static SingletonClass* mInstance;
public:
  static SingletonClass& getInstance()
  {
    if( !mInstance )
    {
      mInstance = new SingletonClass;
      … Initialize the Singleton Class …
    }
    return *mInstance;
  }
};

SingletonClass* SingletonClass::mInstance = NULL;
```

The class is accessed via a static member function

The Singleton pattern

```
class SingletonClass {
private:
  SingletonClass(){}
  static SingletonClass* mInstance;
public:
  static SingletonClass& getInstance()
  {
    if( !mInstance )
    {
      mInstance = new SingletonClass;
      … Initialize the Singleton Class …
    }
    return *mInstance;
  }
};

SingletonClass* SingletonClass::mInstance = NULL;
```

The constructor is only called the first time getInstance() is invoked. If it is never used, no resources are consumed

The Singleton pattern: Caveats

- Care must be taken with Singletons in multithreaded code (mutex locks!)
- Singletons can be (and frequently are) overused and used inappropriately
- When used inappropriately, they can suffer the same problems as global variables (which are evil)

The Template (Hollywood) pattern

What do Hollywood directors say to amateurs?

The Template (Hollywood) pattern

What do Hollywood directors say to amateurs? “Don’t call us, we’ll call you”

The Template (Hollywood) pattern

What do Hollywood directors say to amateurs? “Don’t call us, we’ll call you”

When you first learn to code you start with “Hello World”, where the top-level entity controls program-flow and all function calls come from above.

The Template (Hollywood) pattern

What do Hollywood directors say to amateurs? “Don’t call us, we’ll call you”

When you first learn to code you start with “Hello World”, where the top-level entity controls program-flow and all function calls come from above.

Can very quickly becomes unsustainable in large or complex programmes, especially with multiple developers.

The Template (Hollywood) pattern

What do Hollywood directors say to amateurs? “Don’t call us, we’ll call you”

When you first learn to code you start with “Hello World”, where the top-level entity controls program-flow and all function calls come from above.

Can very quickly becomes unsustainable in large or complex programmes, especially with multiple developers.

Alternative paradigm: control from the bottom up:

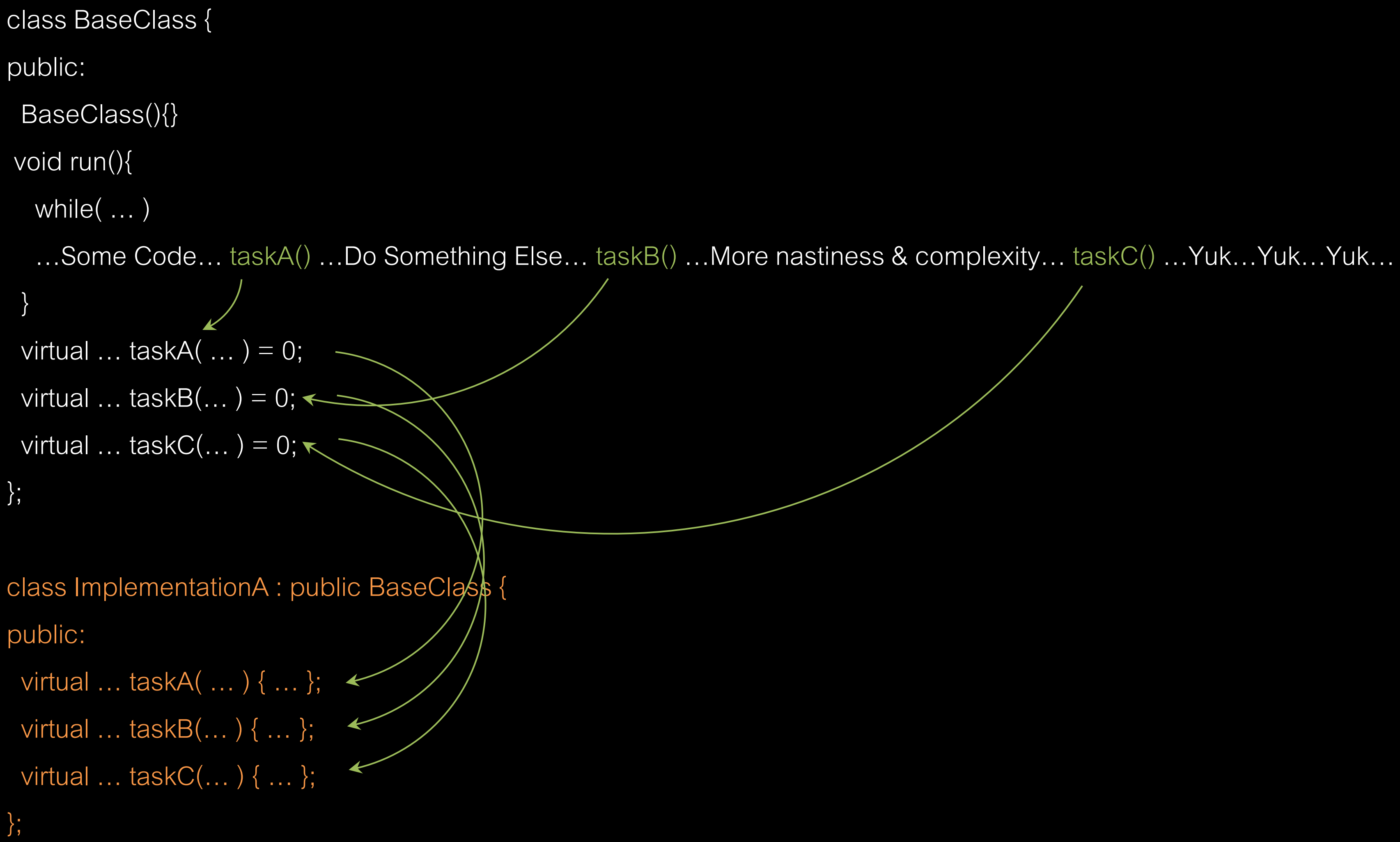
- Divide the program into conceptual steps
- Provide pure virtual functions (“templates”) for each step
- Have the base class control program flow

The Template (Hollywood) pattern

```
class BaseClass {
public:
  BaseClass(){}
 void run(){
    while( … )
    …Some Code… taskA() …Do Something Else… taskB() …More nastiness & complexity… taskC() …Yuk…Yuk…Yuk…
  }
  virtual … taskA( … ) = 0;
  virtual … taskB(… ) = 0;
  virtual … taskC(… ) = 0;
};

class ImplementationA : public BaseClass {
public:
  virtual … taskA( … ) { … };
  virtual … taskB(… ) { … };
  virtual … taskC(… ) { … };
};
```

The Template (Hollywood) pattern



```
class BaseClass {
public:
  BaseClass(){}
 void run(){
    while( … )
     …Some Code… taskA() …Do Something Else… taskB() …More nastiness & complexity… taskC() …Yuk…Yuk…Yuk…
  }
  virtual … taskA( … ) = 0;
  virtual … taskB(… ) = 0;
  virtual … taskC(… ) = 0;
};

class ImplementationA : public BaseClass {
public:
  virtual … taskA( … ) { … };
  virtual … taskB(… ) { … };
  virtual … taskC(… ) { … };
};
```

Object Pool pattern

Some objects are very costly (in time) to instantiate

- Threads
- Large amounts of memory
- Sockets

But may be used frequently, albeit for a very short time

Creating a new object each time would just be stupid

Object Pool pattern

An Object Pool creates the objects outside the time-critical code

Object Pool pattern

An Object Pool creates the objects outside the time-critical code

In the time-critical section, the code takes ownership of an object in the pool, uses it, cleans it and returns it.

Object Pool pattern

An Object Pool creates the objects outside the time-critical code

In the time-critical section, the code takes ownership of an object in the pool, uses it, cleans it and returns it.

If the object is not returned in a clean state

- the next user of the object cannot guarantee the object's behaviour
- there is a security risk (confidential data in a memory)

Object Pool pattern

An Object Pool creates the objects outside the time-critical code

In the time-critical section, the code takes ownership of an object in the pool, uses it, cleans it and returns it.

If the object is not returned in a clean state

- the next user of the object cannot guarantee the object's behaviour
- there is a security risk (confidential data in a memory)

An Object Pool with unclean objects is often called a CESSPOOL

Think plagues and velociraptors…

And finally…

Curiously Recursive Template pattern (CRTP)

Let’s jump straight in with an example

Curiously Recursive Template pattern (CRTP)

Let’s jump straight in with an example

```
template < class T >
class BaseClass {
public:
   …
};

class DerivedClass : public BaseClass< DerivedClass >  {
public:
  …
};
```

Curiously Recursive Template pattern (CRTP)

Let’s jump straight in with an example

```
template < class T >
class BaseClass {
public:
   …
};

class DerivedClass : public BaseClass< DerivedClass >  {
public:
  …
};
```

Note straight-off: This base class cannot be used for polymorphism. Each base class is custom to its derived type.

Curiously Recursive Template pattern (CRTP)

In normal (runtime) polymorphism the base class is unaware of which concrete type it is

In CRTP (also called static or compile-time polymorphism), the base class can do things like:

```
template < class T >
class BaseClass {
public:
  … some_function( … )
   {
     … static_cast<T*>(this) …
     T::static_function();
   }
};
```

The base class can cast itself to the derived type

It can access static members of the derived type

CRTP common use-case

Using runtime polymorphism, if an object is copyable, then every derived type must implement the clone() method, so that the object is copied as the derived type, not the base type.

```
class Shape {
public:
  virtual Shape* clone() = 0;
};

class Circle : public Shape  {
public:
  virtual Shape* clone() { return new Circle( *this ); }
};

class Square : public Shape  {
public:
  virtual Shape* clone() { return new Square( *this ); }
};
```

CRTP common use-case

Using runtime polymorphism, if an object is copyable, then every derived type must implement the clone() method, so that the object is copied as the derived type, not the base type.

```
class Shape {
public:
  virtual Shape* clone() = 0;
};

class Circle : public Shape  {
public:
  virtual Shape* clone() { return new Circle( *this ); }
};
```

Tedious

```
class Square : public Shape  {
public:
  virtual Shape* clone() { return new Square( *this ); }
};
```

Tedious

CRTP common use-case

Using runtime polymorphism, if an object is copyable, then every derived type must implement the clone() method, so that the object is copied as the derived type, not the base type.

```
class Shape {
public:
  virtual Shape* clone() = 0;
};

template < class T >
class ShapeCRTP {
public:
  virtual Shape* clone() return new T( static_cast<T&> ( *this ) );
};

class Circle : public ShapeCRTP< Circle > {};
class Square : public ShapeCRTP< Square > {};
```

Do it once for all derived types

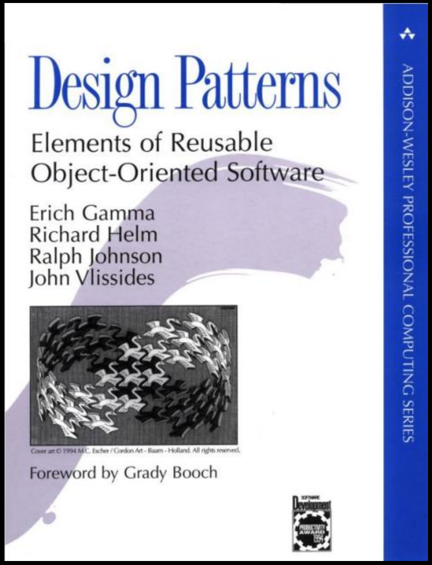
Conclusions

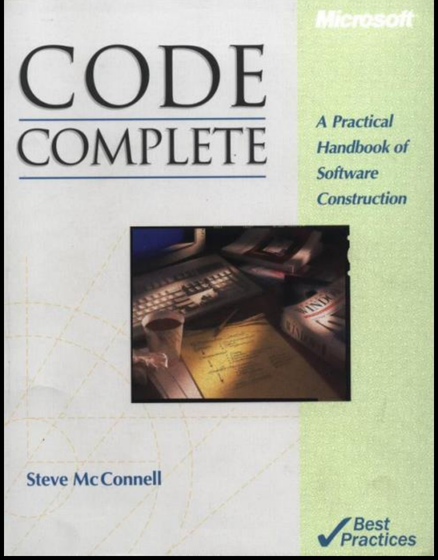
This was just a brief summary of some of the most common and useful design patterns (at least in my experience)

Software design patterns are not magic and they do not solve all of your problems

They do, however, point you to best practice and help you become a better programmer

If you want to be a programmer, rather than someone who codes, read at least one of the following:





Exercise

You have (an arbitrary number of) independent classes and you want to track how many objects of each type are created.

Using CRTP, design a utility class which

- Counts the number of objects created for an arbitrary number of arbitrary classes
- Counts the number of objects which are alive at any particular time
- Adds a static “usage_stats()” function to each class which prints to std::cout a message of the form:

Class ‘*ClassTypeID*’ | *xxx* copies created | *yyy* copies currently alive

Spare

Software Design Patterns: Used in anger

Abstract factory
Builder
Factory method
Lazy initialization
Multiton
Object pool
Prototype
Resource acquisition is initialization
Singleton
Adapter or Wrapper or Translator.
Bridge
Composite
Curiously recursive template pattern
Decorator
Facade
Flyweight
Front Controller
Module
Proxy
Twin

Blackboard
Chain of responsibility
Command
Interpreter
Iterator
Mediator
Memento
Null object
Observer or Publish/subscribe
Servant
Specification
State
Strategy
Template or Hollywood method
Visitor

⋮