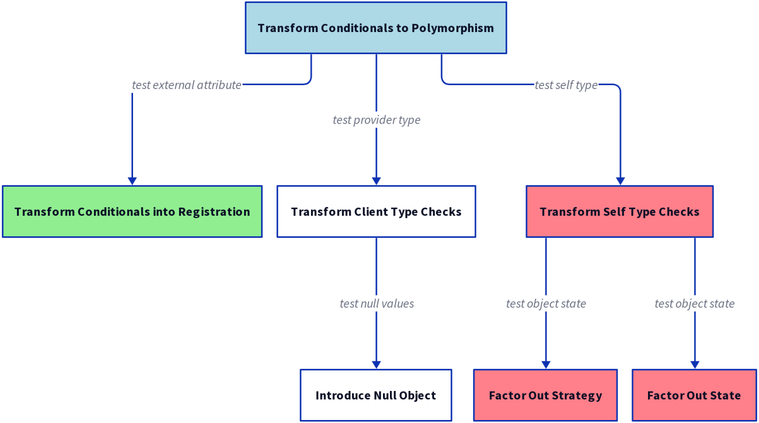
Transform Conditionals to Polymorphism

Software Evolution and Reverse Engineering

Refolli F. 865955

March 2, 2025

Overview of the Cluster



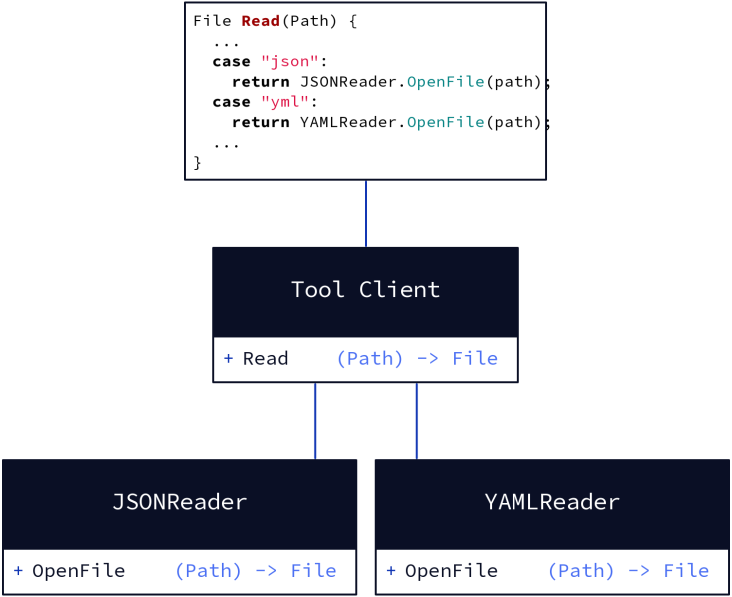
Index

1. External checks
   - Transform Conditionals into Registration
2. Checks on caller
   - Transform Client Type Checks
   - Introduce Null Object
3. Checks on callee
   - Transform Self Type Checks
   - Factor Out Strategy
   - Factor Out State
4. Tradeoffs
5. Tips

Transform Conditionals into Registration

Improve the modularity of a system by replacing conditionals in clients with a registration mechanism.

Before



```
File Read(Path) {
  ...
  case "json":
    return JSONReader.OpenFile(path);
  case "yml":
    return YAMLReader.OpenFile(path);
  ...
}
```

Tool Client

+ Read (Path) -> File

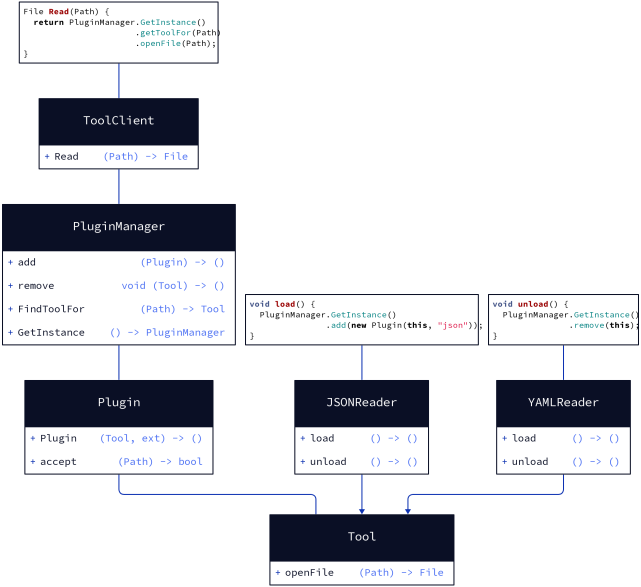
JSONReader

+ OpenFile (Path) -> File

YAMLReader

+ OpenFile (Path) -> File

After



```
File Read(Path) {
  return PluginManager.GetInstance()
                      .getToolFor(Path)
                      .openFile(Path);
}
```

ToolClient

+ Read (Path) -> File

PluginManager

+ add (Plugin) -> ()

+ remove void (Tool) -> ()

+ FindToolFor (Path) -> Tool

+ GetInstance () -> PluginManager

```
void load() {
  PluginManager.GetInstance()
               .add(new Plugin(this, "json"));
}
```

```
void unload() {
  PluginManager.GetInstance()
               .remove(this);
}
```

Plugin

+ Plugin (Tool, ext) -> ()

+ accept (Path) -> bool

JSONReader

+ load () -> ()

+ unload () -> ()

YAMLReader

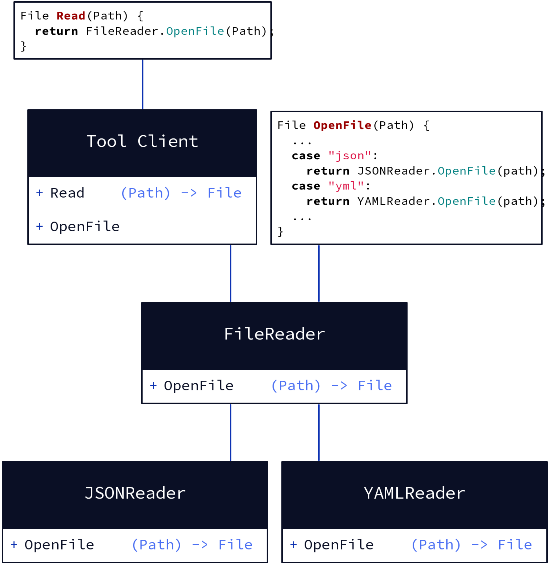
+ load () -> ()

+ unload () -> ()

Tool

+ openFile (Path) -> File

KISS



```
File Read(Path) {
  return FileReader.OpenFile(Path);
}
```

```
Tool Client
+ Read      (Path) -> File
+ OpenFile
```

```
File OpenFile(Path) {
  ...
  case "json":
    return JSONReader.OpenFile(path);
  case "yml":
    return YAMLReader.OpenFile(path);
  ...
}
```

```
FileReader
+ OpenFile     (Path) -> File
```

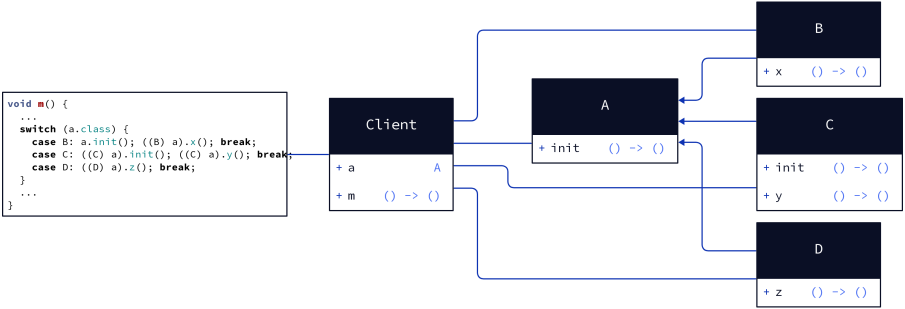
```
JSONReader
+ OpenFile     (Path) -> File
```

```
YAMLReader
+ OpenFile     (Path) -> File
```

Transform Client Type Checks

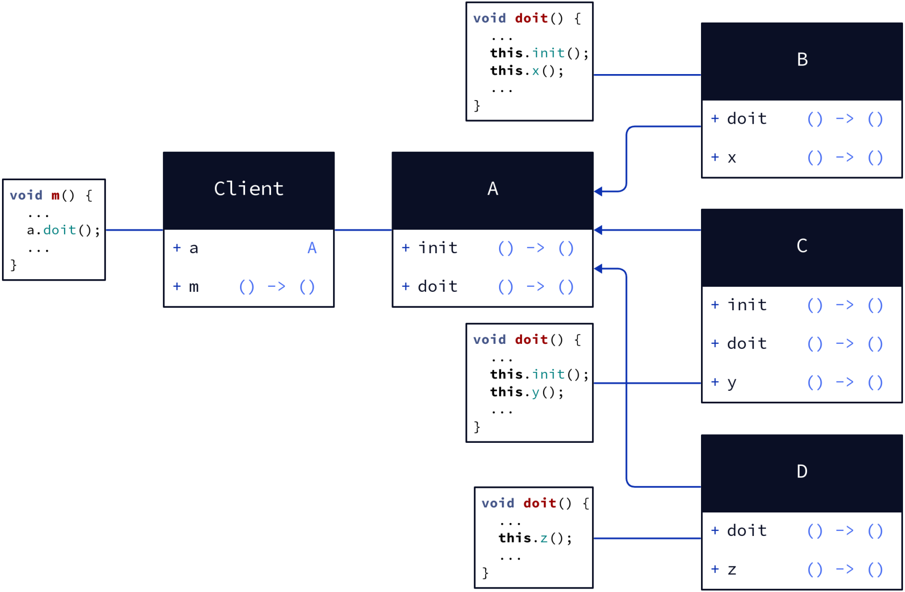
Reduce client/provider coupling by transforming conditional code that tests the type of the provider into a polymorphic call to a new provider method.

Before



```
void m() {
  ...
  switch (a.class) {
    case B: a.init(); ((B) a).x(); break;
    case C: ((C) a).init(); ((C) a).y(); break;
    case D: ((D) a).z(); break;
  }
  ...
}
```

After



```
void m() {
  ...
  a.doit();
  ...
}
```

Client

+ a A

+ m () -> ()

A

+ init () -> ()

+ doit () -> ()

```
void doit() {
  ...
  this.init();
  this.x();
  ...
}
```

B

+ doit () -> ()

+ x () -> ()

C

+ init () -> ()

+ doit () -> ()

+ y () -> ()

```
void doit() {
  ...
  this.init();
  this.y();
  ...
}
```

D

+ doit () -> ()

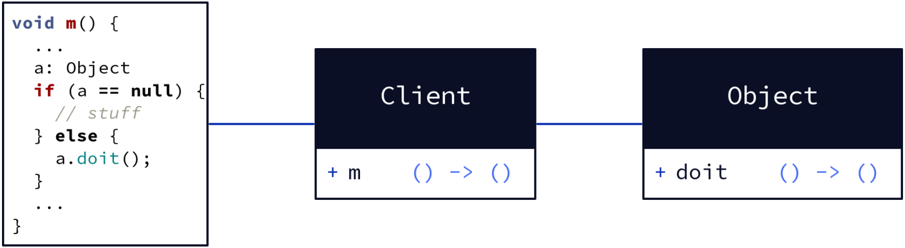
+ z () -> ()

```
void doit() {
  ...
  this.z();
  ...
}
```

Introduce Null Object

Eliminate conditional code that tests for null values by applying the Null Object design pattern.

Before



```
void m() {
  ...
  a: Object
  if (a == null) {
    // stuff
  } else {
    a.doit();
  }
  ...
}
```

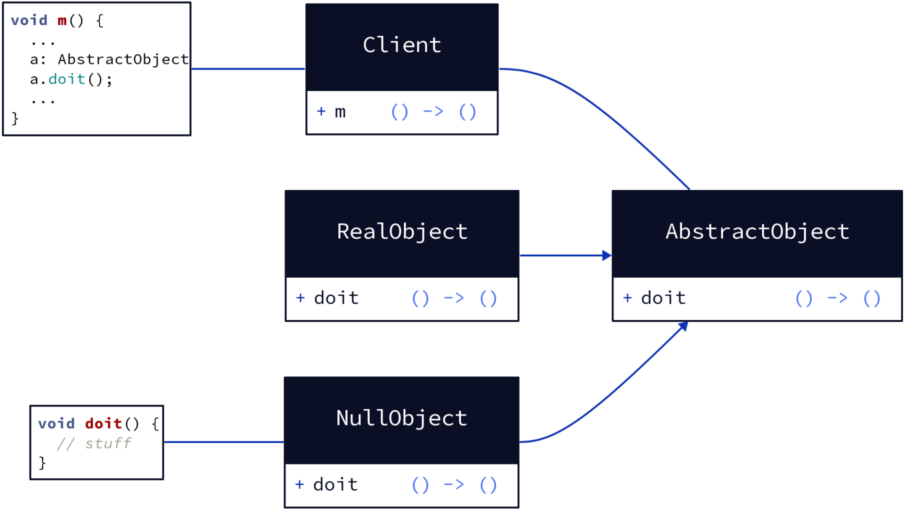
Client

+ m () -> ()

Object

+ doit () -> ()

After



```
void m() {
  ...
  a: AbstractObject
  a.doit();
  ...
}
```

Client

+ m () -> ()

RealObject

+ doit () -> ()

AbstractObject

+ doit () -> ()

NullObject

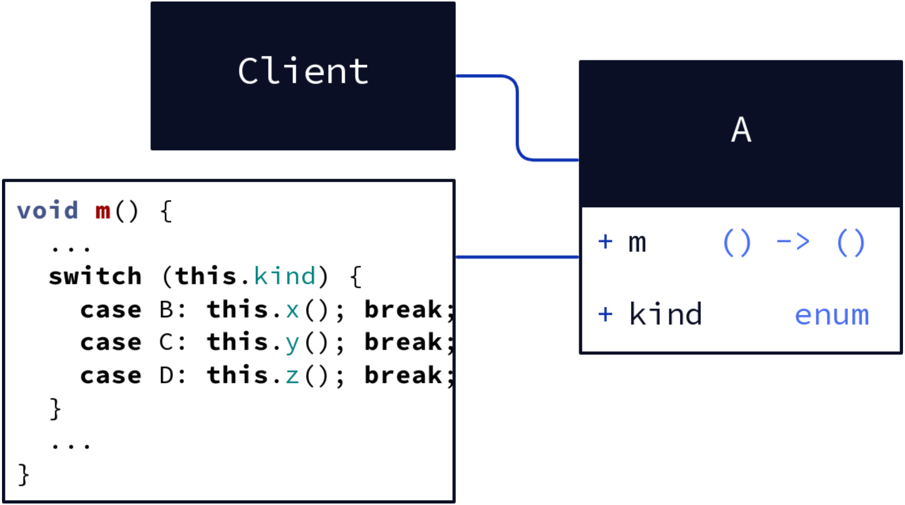
+ doit () -> ()

```
void doit() {
  // stuff
}
```

Transform Self Type Checks

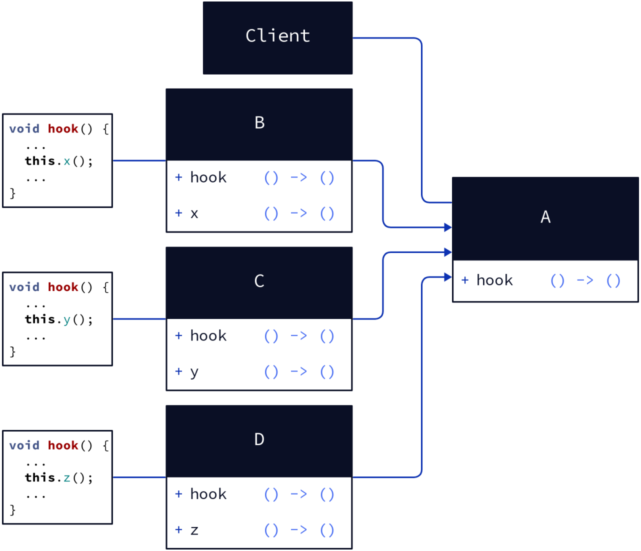
Improve the extensibility of a class by replacing a complex conditional statement with a call to a hook method implemented by subclasses.

Before



```
void m() {
  ...
  switch (this.kind) {
    case B: this.x(); break;
    case C: this.y(); break;
    case D: this.z(); break;
  }
  ...
}
```

After



Client

```
void hook() {
   ...
   this.x();
   ...
}
```

B

+ hook () -> ()

+ x () -> ()

```
void hook() {
   ...
   this.y();
   ...
}
```

C

+ hook () -> ()

+ y () -> ()

```
void hook() {
   ...
   this.z();
   ...
}
```

D

+ hook () -> ()

+ z () -> ()

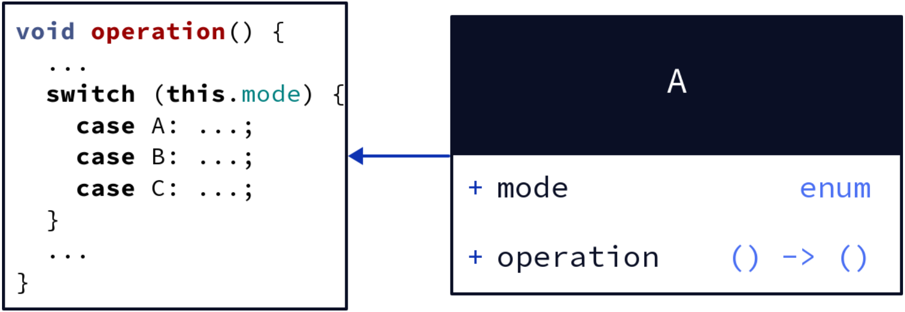
A

+ hook () -> ()

Factor Out Strategy

Eliminate conditional code that selects a suitable algorithm by applying the Strategy design pattern.

Before



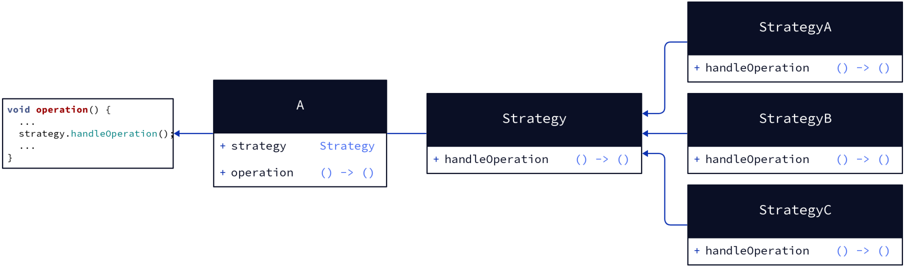
```
void operation() {
  ...
  switch (this.mode) {
    case A: ...;
    case B: ...;
    case C: ...;
  }
  ...
}
```

A

+ mode enum

+ operation () -> ()

After



```
void operation() {
  ...
  strategy.handleOperation();
  ...
}
```

A

+ strategy Strategy

+ operation () -> ()

Strategy

+ handleOperation () -> ()

StrategyA

+ handleOperation () -> ()

StrategyB

+ handleOperation () -> ()

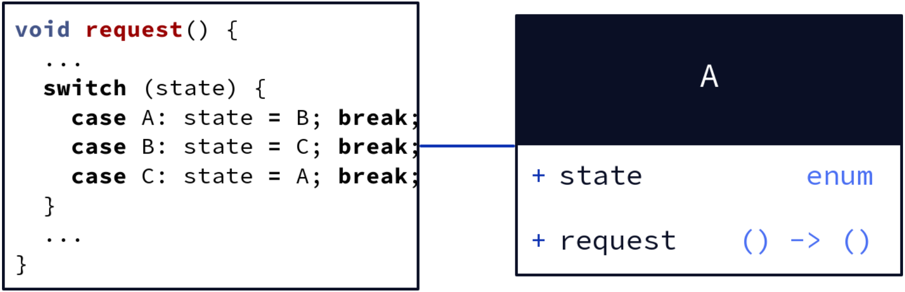
StrategyC

+ handleOperation () -> ()

Factor Out State

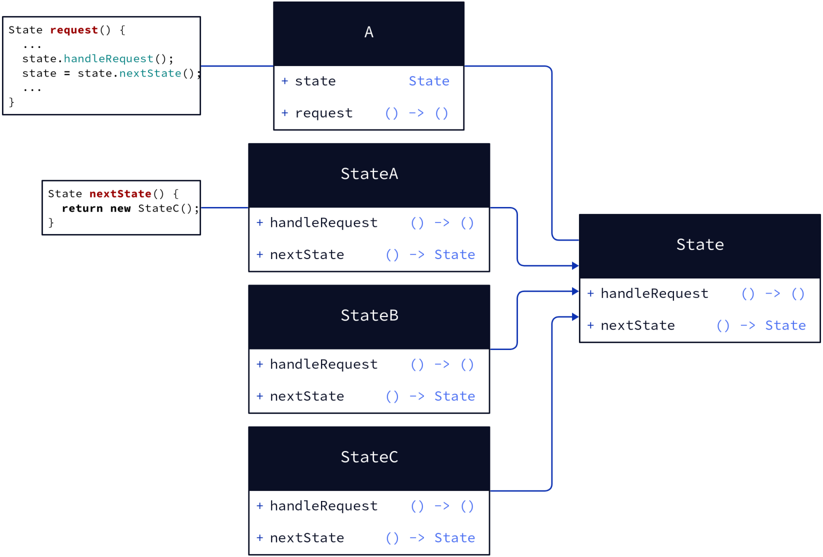
Eliminate complex conditional code over an object's state by applying the State design pattern.

Before



```
void request() {
  ...
  switch (state) {
    case A: state = B; break;
    case B: state = C; break;
    case C: state = A; break;
  }
  ...
}
```

After



```
State request() {
  ...
  state.handleRequest();
  state = state.nextState();
  ...
}
```

A

+ state State

+ request () -> ()

```
State nextState() {
  return new StateC();
}
```

StateA

+ handleRequest () -> ()

+ nextState () -> State

StateB

+ handleRequest () -> ()

+ nextState () -> State

StateC

+ handleRequest () -> ()

+ nextState () -> State

State

+ handleRequest () -> ()

+ nextState () -> State

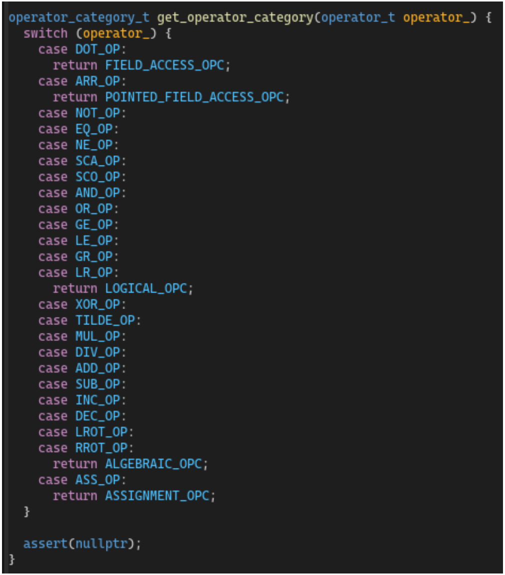
Tradeoffs

- Pros
  - Component behavior isolation.
  - Changes/additions/removal of behavior don't affect (significantly) the clients.
  - Behaviors share a common interface.
- Cons
  - Difficult to get a large picture of the behavior of the subsystem.
  - Explosion of abstractions and classes.
  - Manipulation of class instances is heavily bloated (huge number of instances and GC workload).
  - Sometimes classes are just a cool namespace mechanism.

Tips

1. Explicit checks are not always a problem (it depends on the context), and often can be tolerated / optimal when the number of places in which are done is low or the number of cases is fixed and low.

Tips

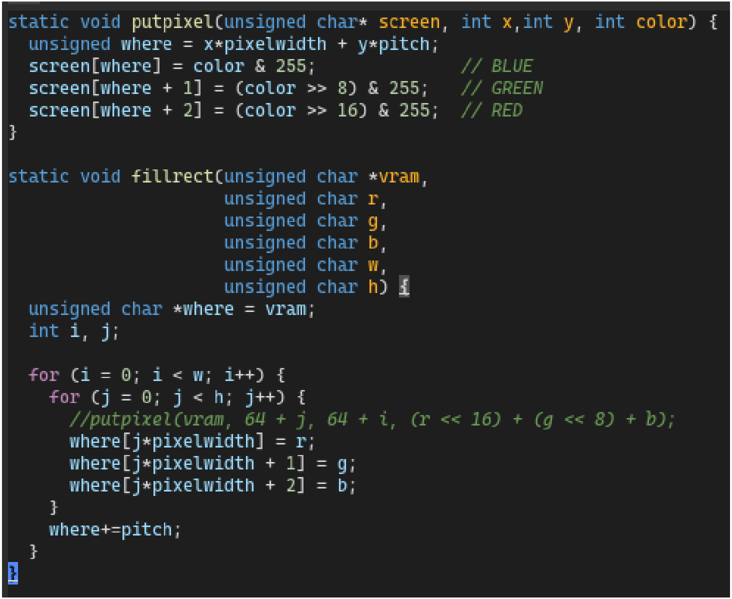


```
operator_category_t get_operator_category(operator_t operator_) {
  switch (operator_) {
    case DOT_OP:
      return FIELD_ACCESS_OPC;
    case ARR_OP:
      return POINTED_FIELD_ACCESS_OPC;
    case NOT_OP:
    case EQ_OP:
    case NE_OP:
    case SCA_OP:
    case SCO_OP:
    case AND_OP:
    case OR_OP:
    case GE_OP:
    case LE_OP:
    case GR_OP:
    case LR_OP:
      return LOGICAL_OPC;
    case XOR_OP:
    case TILDE_OP:
    case MUL_OP:
    case DIV_OP:
    case ADD_OP:
    case SUB_OP:
    case INC_OP:
    case DEC_OP:
    case LROT_OP:
    case RROT_OP:
      return ALGEBRAIC_OPC;
    case ASS_OP:
      return ASSIGNMENT_OPC;
  }

  assert(nullptr);
}
```

Tips

1. Explicit checks are not always a problem (it depends on the context), and often can be tolerated / optimal when the number of places in which are done is low or the number of cases is fixed and low.
2. In some circumstances abstractions are application killers.

Tips

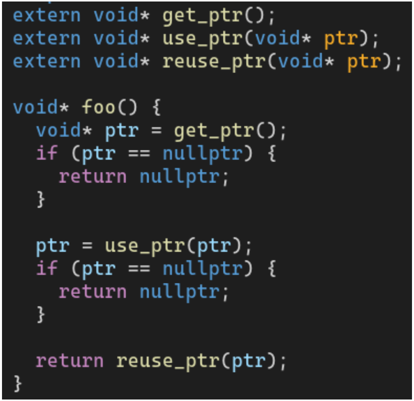


```c
static void putpixel(unsigned char* screen, int x,int y, int color) {
  unsigned where = x*pixelwidth + y*pitch;
  screen[where] = color & 255;              // BLUE
  screen[where + 1] = (color >> 8) & 255;   // GREEN
  screen[where + 2] = (color >> 16) & 255;  // RED
}

static void fillrect(unsigned char *vram,
                     unsigned char r,
                     unsigned char g,
                     unsigned char b,
                     unsigned char w,
                     unsigned char h) {
  unsigned char *where = vram;
  int i, j;

  for (i = 0; i < w; i++) {
    for (j = 0; j < h; j++) {
      //putpixel(vram, 64 + j, 64 + i, (r << 16) + (g << 8) + b);
      where[j*pixelwidth] = r;
      where[j*pixelwidth + 1] = g;
      where[j*pixelwidth + 2] = b;
    }
    where+=pitch;
  }
}
```

Tips

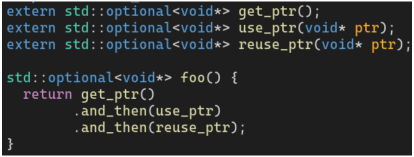
1. Explicit checks are not always a problem (it depends on the context), and often can be tolerated / optimal when the number of places in which are done is low or the number of cases is fixed and low.
2. In some circumstances abstractions are application killers.
3. Most of the time a monad is what you wanted.

Tips



```
extern void* get_ptr();
extern void* use_ptr(void* ptr);
extern void* reuse_ptr(void* ptr);

void* foo() {
  void* ptr = get_ptr();
  if (ptr == nullptr) {
    return nullptr;
  }

  ptr = use_ptr(ptr);
  if (ptr == nullptr) {
    return nullptr;
  }

  return reuse_ptr(ptr);
}
```

Figure: with null checks



```
extern std::optional<void*> get_ptr();
extern std::optional<void*> use_ptr(void* ptr);
extern std::optional<void*> reuse_ptr(void* ptr);

std::optional<void*> foo() {
  return get_ptr()
         .and_then(use_ptr)
         .and_then(reuse_ptr);
}
```

Figure: with monads

The End