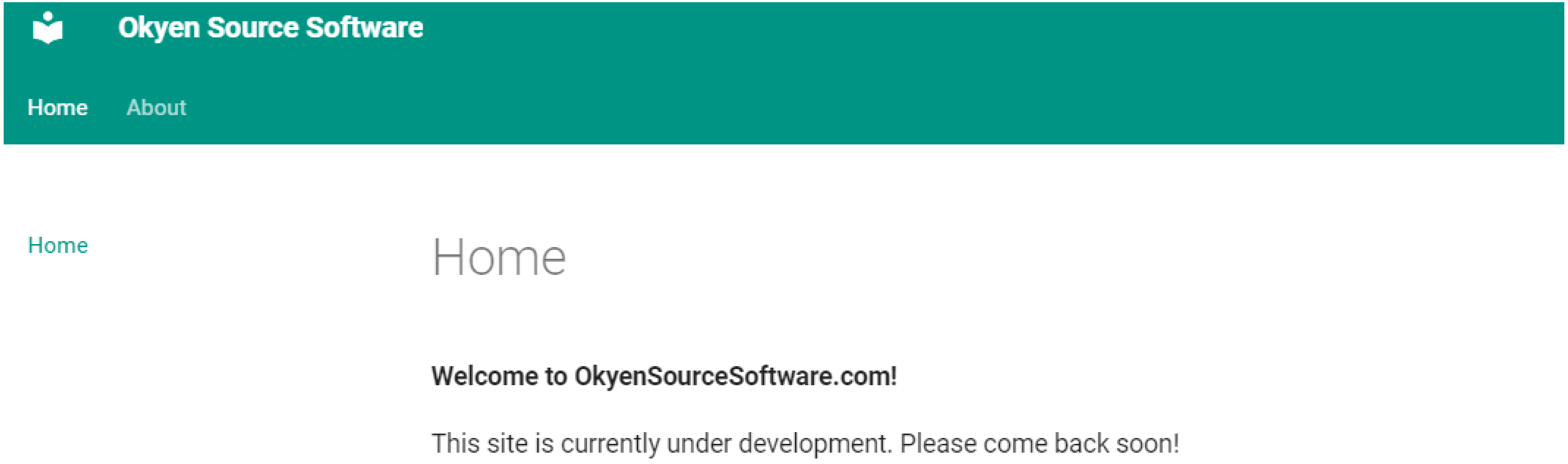
Developing Better Code by Isolating Decisions

Michael Okyen
Staff Software Engineer
Plexus

WIP: okyensourcesoftware.com



Today’s roadmap

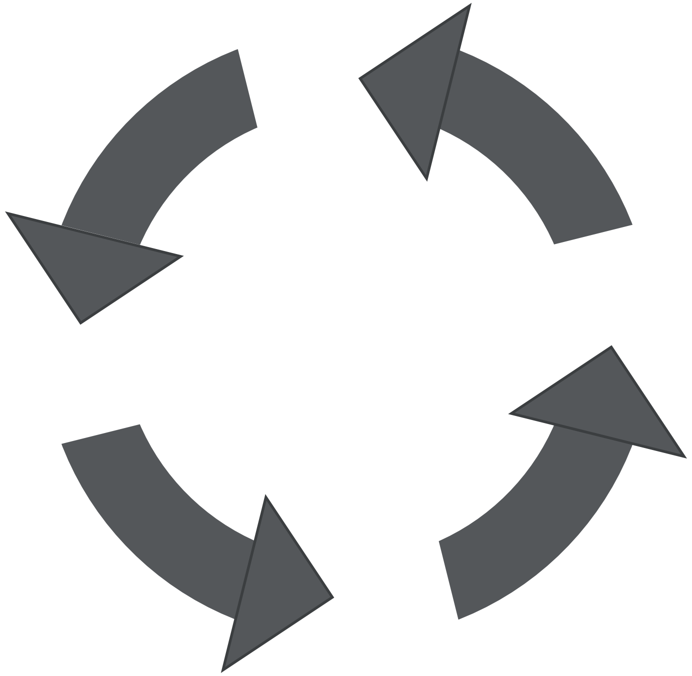
- Challenges with software design
- Decision-Making Isolation overview
- Invariance in entity design
- Unit testing in DMI

COLLEGE OF ENGINEERING
MECHANICAL ENGINEERING
VIRGINIA TECH®

Mech → Robots → Software

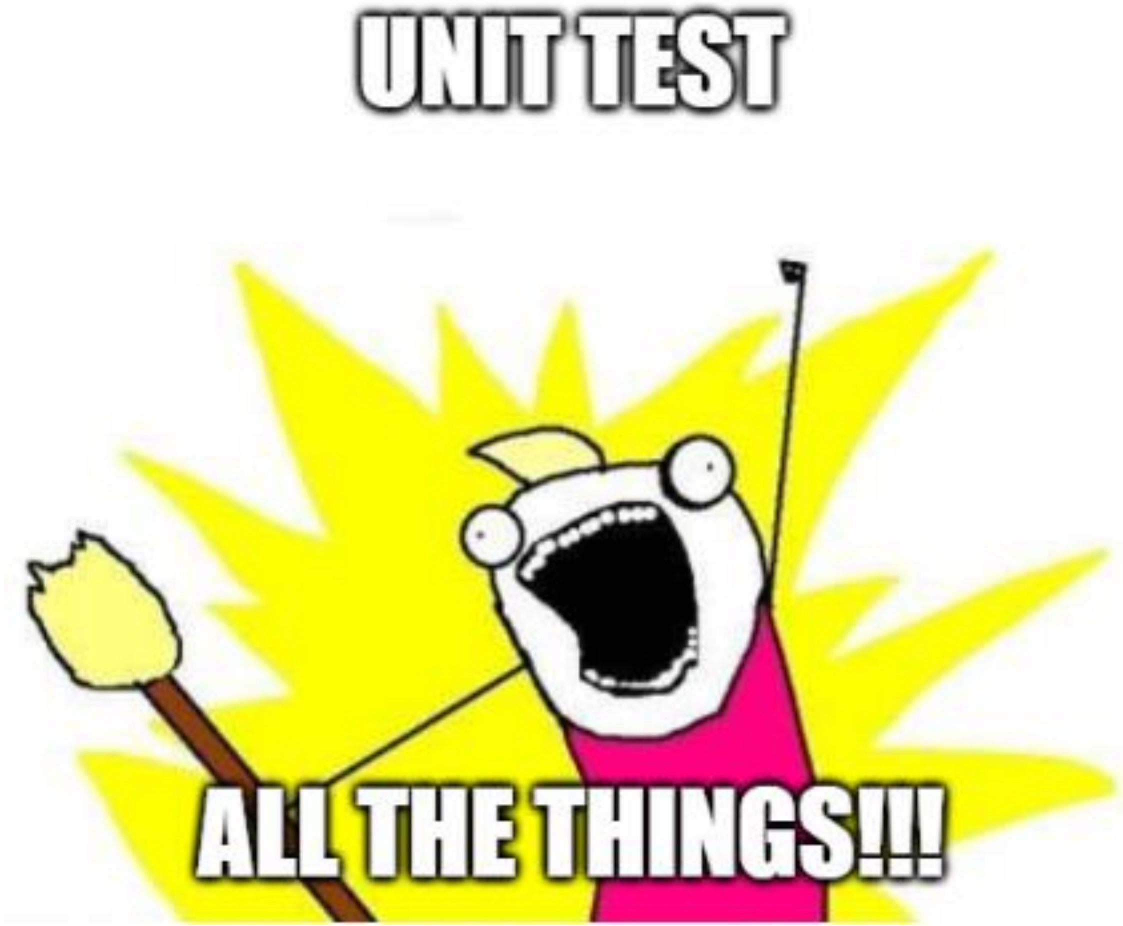
A B

Cycle of Quality



UNIT TESTING

My old unit testing strategy



We needed to change the unit testing focus.

Revised focus

To be skilled at utilizing unit testing, we need to focus on getting better at **choosing what code to test** and **designing code that is testable** rather than getting better at *writing more complex tests*.

CAUTION

WORK

IN PROGRESS

“High-quality software is easy to change, easy to extend, and easy to test.”

– Klaus Iglberger, 2022

Today’s definition - Quality software is easy to:

Read

Modify

Extend

Test

What are some acronyms we use when discussing design?

What are some acronyms we use when discussing design?

SOLID

DRY

KISS

YAGNI

DAMP (Descriptive And Meaningful Phrases)

GRASP

CUPID

SOLID Principles

S Single Responsibility Principle
O Open-Closed Principle
L Liskov Substitution Principle
I Interface Segregation Principle
D Dependency Inversion Principle

S&O Definitions (from Robert Martin)

SRP

“There should never be more than one reason for a class to change.”

“A module should be responsible to one, and only one, actor.” (From *Clean Architecture*, 2017)

OCP

“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.” (From *Clean Code*, 2008)

Who felt like they understood SRP and OCP when you first read them?

Who later felt that their initial understanding was wrong?

Who feels that it’s still hard to follow S&O?

What makes S&O something that makes implementing these concepts hard?

SRP

“There should never be more than one reason for a class to change.”

“A module should be responsible to one, and only one, actor.” (From *Clean Architecture*, 2017)

OCP

“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.” (From *Clean Code*, 2008)

What makes S&O something that makes implementing these concepts hard?

SRP

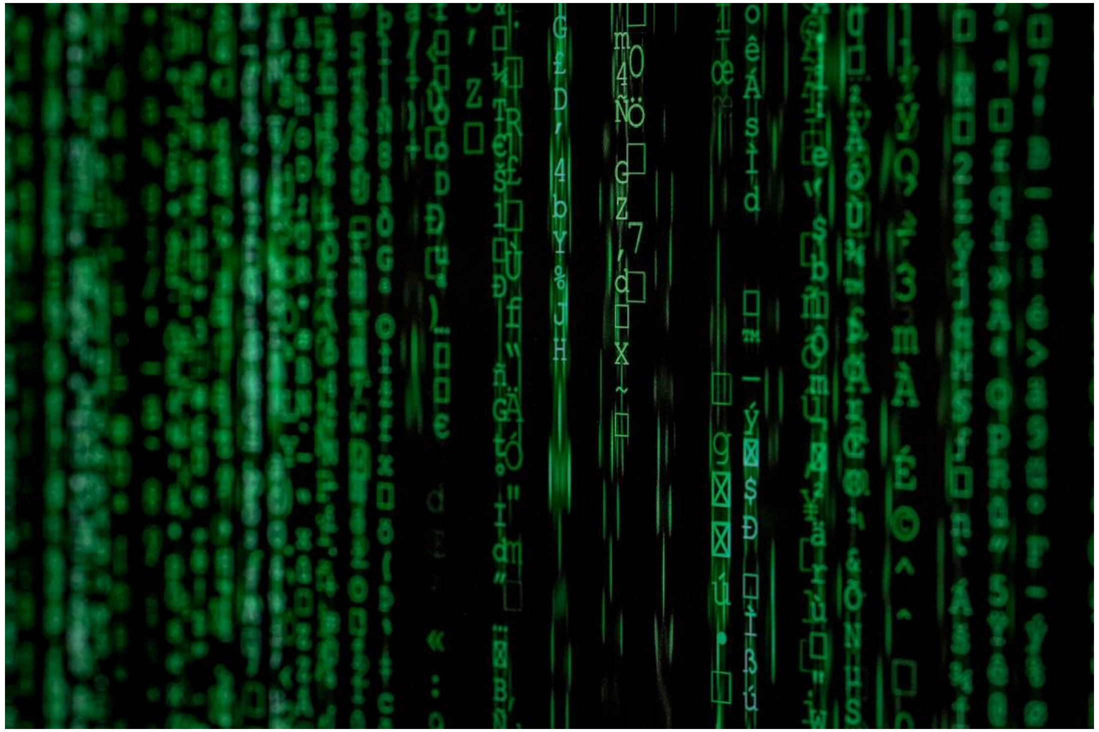
"There should never be more than **one reason** for a class to change."

"A module should be responsible to **one, and only one, actor**." (From *Clean Architecture*, 2017)

OCP

"Software entities (classes, modules, functions, etc.) should be open for extension, but **closed** for modification." (From *Clean Code*, 2008)

Design can be like raking leaves



How do you decide when you've reduced an entity to a *single* responsibility?

How do you decide when you’ve done enough unit testing?

How do you help new or inexperienced teammates adhere to these standards?

DMI
Decision-Making Isolation

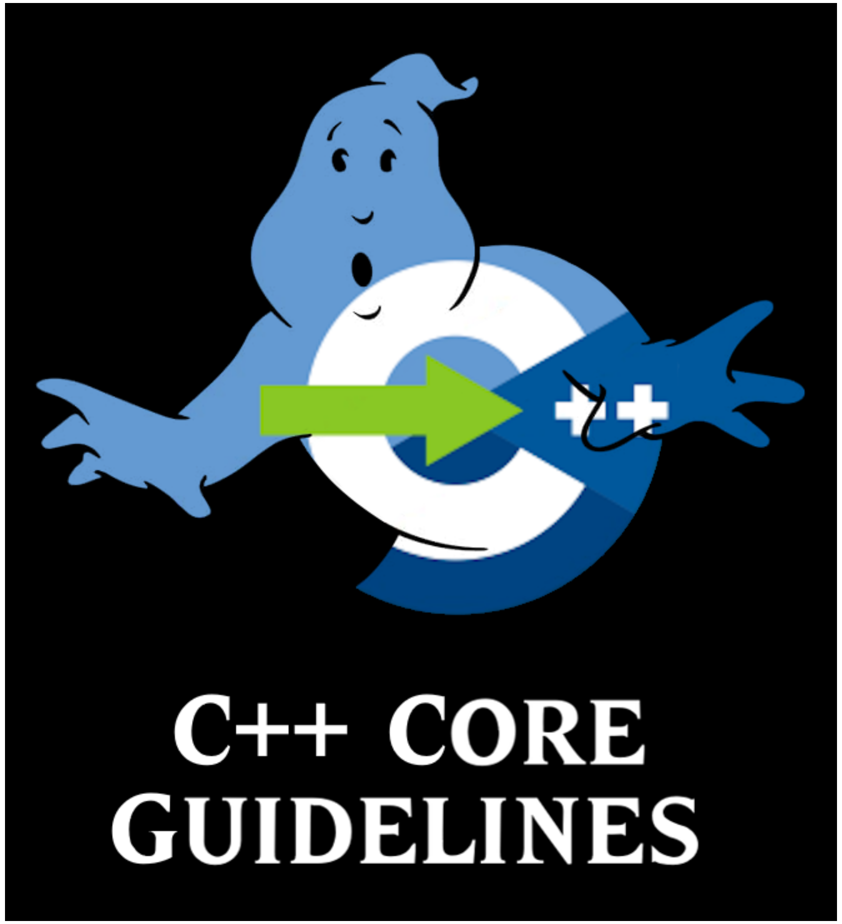
Vision

The most important parts of our code are the places where we decide what should happen.

If those key decisions are implemented correctly, then we can feel confident that our code is trying to do what we expect it to do.

If these decisions are isolated, that critical code is easy to read, modify, extend, and test.

Need help? Who you gonna call?



C++ Core Guidelines

What is Decision-Making Isolation?

Separating decision-making into pure free functions or small classes without external dependencies. 1,5

Defaulting to using free functions and publicly-available data, such as structs or variables, instead of classes. Only use a class if the data has invariance or is part of a component boundary. 1,2,3,4

Unit testing all decision-making code thoroughly. 5

C++ Core Guidelines:

1 - CG F.8   2 - CG C.2   3 - CG C.4   4 - CG C.8   5 - CG A.1

Before DMI

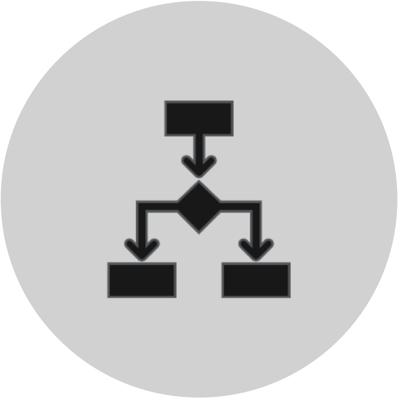
```
class RequestHandler {
private:
    Outputter* m_outputter;
public:
    RequestHandler (Outputter* outputter) : m_outputter(outputter)
    {}

    void processRequest(EState state){
        switch (state){
            case EState::State1:
                m_outputter->send((int)EData::Data1);
                break;
            case EState::State2:
                m_outputter->send(2);
                break;
        }
    }
};
```

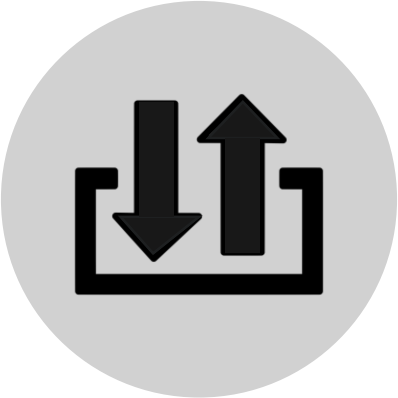
SW Quality:
Readability
Modifiability
Extensibility
Testability

S&O
SRP
OCP

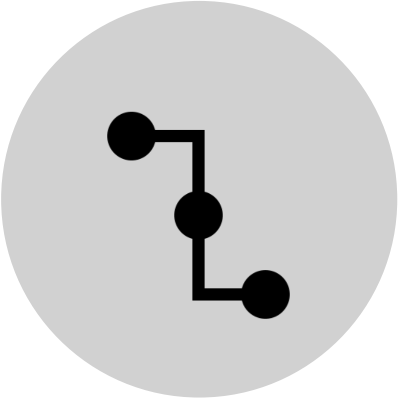
Three types of code



DECISION-MAKING



IO



WIRING

Refactored with DMI

```
class RequestHandler {
private:
    Outputter* m_outputter;
public:
    RequestHandler (Outputter* outputter) : m_outputter(outputter) {}
    //Wiring
    void processRequest_wiring(EState state) {
        auto data{RequestHandlerHelpers::decide(state)}; //Decision-making
        m_outputter->send(data); // I/O
    }};

namespace RequestHandlerHelpers {
int decide(EState state) {
    int ret;
    switch (state) {
        case EState::State1:
            ret = (int)EData::Data1;
            break;
        case EState::State2:
            ret = 2;
            break;
    }
        return ret;
}}
```

SW Quality:
Readability
Modifiability
Extensibility
Testability

S&O
SRP
OCP

Refactored with DMI

```
class RequestHandler {
private:
    Outputter* m_outputter;
public:
    RequestHandler (Outputter* outputter) : m_outputter(outputter) {}
    //Wiring
    void processRequest_wiring(EState state) {
        auto data{RequestHandlerHelpers::decide(state)};        Decision-making
        m_outputter->send(data); // I/O
    }};

namespace RequestHandlerHelpers {
int decide(EState state) {                Decision-making
    int ret;
    switch (state) {
        case EState::State1:
            ret = (int)EData::Data1;
            break;
        case EState::State2:
            ret = 2;
            break;
    }
        return ret;
}}
```

SW Quality:
Readability
Modifiability
Extensibility
Testability

S&O
SRP
OCP

Refactored with DMI

```
class RequestHandler {
private:
    Outputter* m_outputter;
public:
    RequestHandler (Outputter* outputter) : m_outputter(outputter) {}
    //Wiring
    void processRequest_wiring(EState state) {
        auto data{RequestHandlerHelpers::decide(state)}; //Decision-making
        m_outputter->send(data);                                          IO
    }};

namespace RequestHandlerHelpers {
int decide(EState state) {
    int ret;
    switch (state) {
        case EState::State1:
            ret = (int)EData::Data1;
            break;
        case EState::State2:
            ret = 2;
            break;
    }
        return ret;
}}
```

SW Quality:
Readability
Modifiability
Extensibility
Testability

S&O
SRP
OCP

Refactored with DMI

```
class RequestHandler {
private:
    Outputter* m_outputter;
public:
    RequestHandler (Outputter* outputter) : m_outputter(outputter) {}
    //Wiring
    void processRequest_wiring(EState state) {
        auto data{RequestHandlerHelpers::decide(state)}; //Decision-making
        m_outputter->send(data); // I/O
    }};
```

Wiring

```
namespace RequestHandlerHelpers {
int decide(EState state) {
    int ret;
    switch (state) {
        case EState::State1:
            ret = (int)EData::Data1;
            break;
        case EState::State2:
            ret = 2;
            break;
    }
        return ret;
}}
```

SW Quality:
Readability
Modifiability
Extensibility
Testability

S&O
SRP
OCP

Before

```
class RequestHandler {
private:
    Outputter* m_outputter;
public:
    RequestHandler (Outputter* outputter) :
        m_outputter(outputter)
    {}

   void processRequest(EState state){
        switch (state){
            case EState::State1:
                m_outputter->send((int)EData::Data1);
                break;
            case EState::State2:
                m_outputter->send(2);
                break;
        }
    }
};
```

After

```
class RequestHandler {
private:
    Outputter* m_outputter;
public:
    RequestHandler (Outputter* outputter) :
        m_outputter(outputter) {}
    //Wiring
    void processRequest_wiring(EState state) {
        //Decision-making
        auto data{RequestHandlerHelpers::decide(state)};
        m_outputter->send(data); // I/O
    }};
namespace RequestHandlerHelpers {
int decide(EState state) {
    int ret;
    switch (state) {
        case EState::State1:
            ret = (int)EData::Data1;
            break;
        case EState::State2:
            ret = 2;
            break;
    }
        return ret;
}}
```

How to divide code into decision-making, IO, and wiring

What is Decision-Making Isolation?

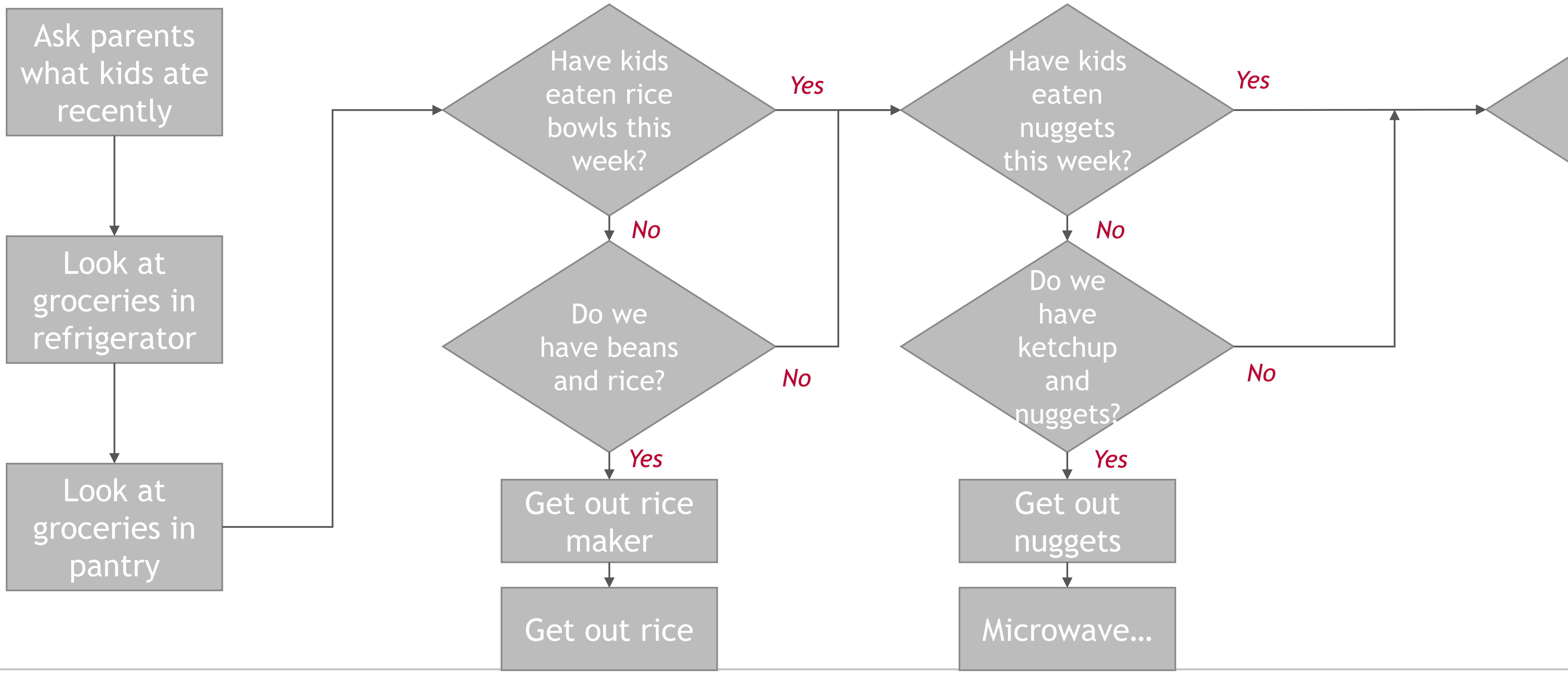
Separating decision-making into pure free functions or small classes without external dependencies.

Defaulting to using free functions and publicly-available data, such as structs or variables, instead of classes. Only use a class if the data has invariance or is part of a component boundary.

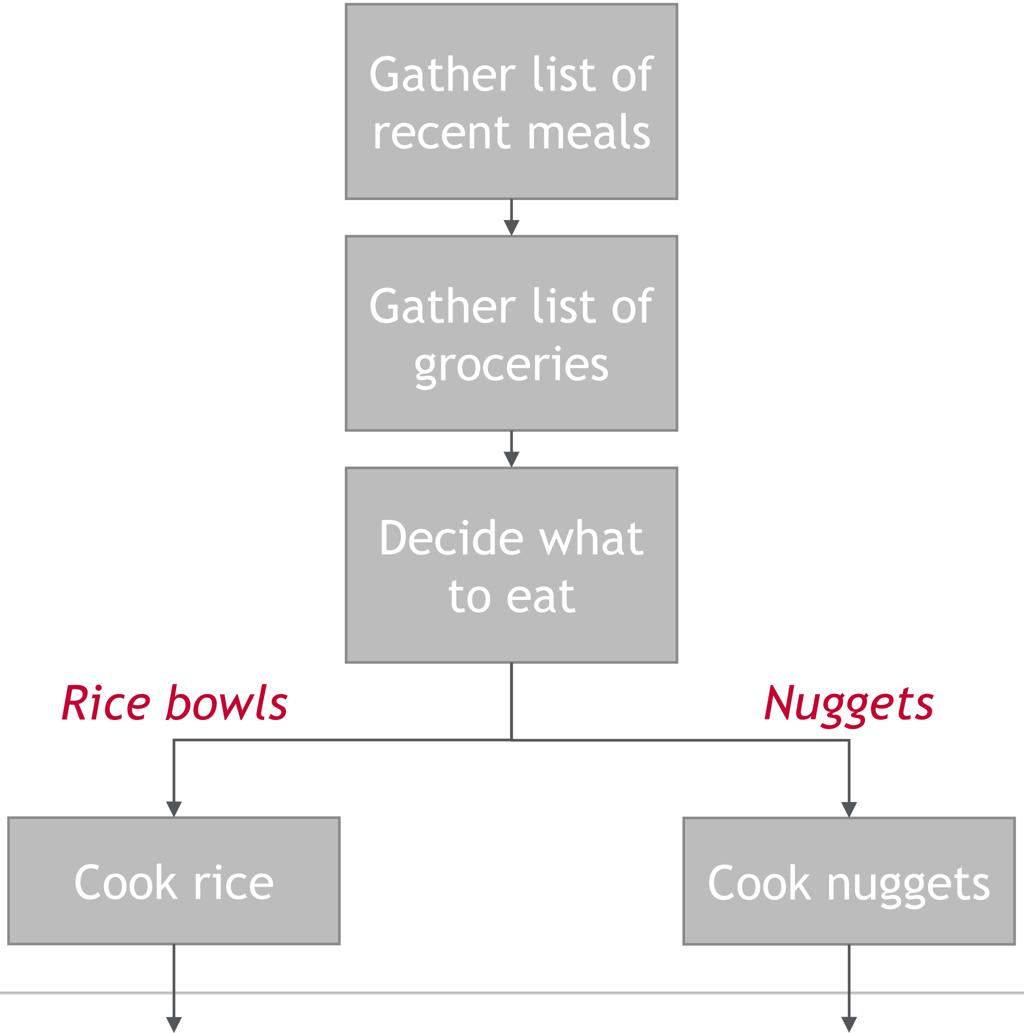
Unit testing all decision-making code thoroughly.

Picky Eaters

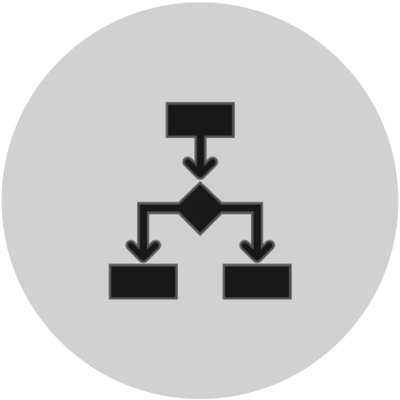
Choosing what to eat for dinner and making it



Choosing what to eat for dinner and making it



Decision-making



Working definition: Decision-making

Determines what the code should do

Decision-making attributes

- Focused on conditionals, non-trivial computations, or branching structures (if, switch, etc., potentially for loops).

- Often returns flags or enums that indicate what actions should be taken.

- Can be algorithms.

Decision-making - return enum

```
EPizzaType determinePizzaToOrder(bool isWifePresent,
                                 bool isKid1Present,
                                 bool isKid2Present) {
    if (isKid1Present) {
        return EPizzaType::Cheese;
    } else if (isWifePresent && isKid2Present) {
        return EPizzaType::Sausage;
    } else if (isWifePresent) {
        return EPizzaType::Hawaiian;
    } else if (isKid2Present) {
        return EPizzaType::Pepperoni;
    } else {
        return EPizzaType::Veggie;
    }
}
```

Decision-making - return bool

```
bool shouldWeGoOutToEat(int energyLevel,
                        unsigned int numKidsPresent,
                        EDayOfWeek day) {
    if (numKidsPresent == 0) {
        return true;
    } else if ( (energyLevel > MIN_PATIENCE_ENERGY_THRESHOLD) &&
                (isWeekend(day)) ) {
        return true;
    } else {
        return false
    }
}
```

Decision-making - algorithm

```
int calculateEnergyLevel(float rollingAvgHoursSleep,
                         unsigned int numCupsCaffeine) {
    auto baselineEnergyLevel{0.0863 * std::pow(rollingAvgHoursSleep, 3.352)};
    auto caffeineMultiplier{1 + 0.3 * (float)numCupsCaffeine};
    return  (int)(baselineEnergyLevel * caffeineMultiplier);
}
```

Pure free functions and small classes are easier to make **constexpr**

IO

Working definition: IO

Gathers or supplies data for decision-making or

does something with the outputs of the decisions.

Types of IO

- Interacting with external code like UIs, HW, databases, messaging protocols.
- User-defined entity from a different component.
- Direct calls made in the wiring.

IO examples

```
class HealthMonitor {
public:
  void performSystemHealthCheck() {
    auto prevAreAllSubsHealthy{m_areAllSubsHealthy};
    m_areAllSubsHealthy = HealthMonitorHelpers::checkAreSubsHealthy(m_subSysHealthTracker);

     auto currentTimeRaw{std::chrono::steady_clock::now().time_since_epoch()};
    auto currentTimeSinceBoot{std::chrono::duration_cast<std::chrono::milliseconds>(currentTimeRaw)};
    auto timeSinceSubsUnhealthy{currentTimeSinceBoot - m_timeSubsBecameUnhealthy};

     EActionToTake action{HealthMonitorHelpers::getAction(prevAreAllSubsHealthy, m_areAllSubsHealthy,
                                                                   timeSinceSubsUnhealthy, m_timeSubsBecameUnhealthy)};

      if (action == EActionsToTake::NotifyWarning) {m_warningNotifier->notifyHealthWarning()};
     else if (action == EActionsToTake::UpdateErrorTime) {m_timeSubsBecameUnhealthy = currentTimeSinceBoot};
  }
private:
  SubsystemHealthTracker* m_subSysHealthTracker;
  WarningNotifier* m_warningNotifier;
  bool m_areAllSubsHealthy{false};
  std::chrono::milliseconds m_timeSubsBecameUnhealthy;
  constexpr std::chrono::milliseconds maxSubUnhealthyTime{100};
};
```

Wiring

Working definition: Wiring

Stitches together the decision-making and IO at a higher level of abstraction so it is easy to read and understand.

Wiring attributes

- Describes how data is gathered, decisions are made, and action is taken.
- Boring, easy to read and code review.
- Often difficult to unit test due to IO dependencies.
  - Low value to unit test, so we don't.

“Clean code is simple and direct. Clean code reads like **well-written prose**. Clean code never obscures the designer's intent…”

– Grady Booch

From *Clean Code*, 2008

Wiring simplified

```
class Wiring {
function doStuff() {
  auto a{input->gatherData()};
  auto decision{DecisionMaking::decide(a)};
  if (decision == Option1) {
     writer->write(1);
  } else
     writer->write(2);
  }
}
```

Choosing classes vs structs: invariance

What is Decision-Making Isolation?

Separating decision-making into pure free functions or small classes without external dependencies.

Defaulting to using free functions and publicly-available data, such as structs or variables, instead of classes. Only use a class if the data has invariance or is part of a component boundary.

Unit testing all decision-making code thoroughly.

Aren’t classes and structs almost the same?

CG C.8: Use ‘class’ rather than ‘struct’ if any member is non-public.

Reason:

Readability. To make it clear that something is being hidden/abstracted. This is a useful convention.

CG C.2: Use ‘class’ if the class has an invariant; use ‘struct’ if the data members can vary independently.

Reason:

Readability. Ease of comprehension. The use of class alerts the programmer to the need for an invariant. This is a useful convention.

Defining ‘invariance’ in C++ Core Guidelines

C.2: An invariant is a logical condition for the members of an object that a constructor must establish for the public member functions to assume. After the invariant is established (typically by a constructor) every member function can be called for the object. An invariant can be stated informally (e.g., in a comment) or more formally using Expects.

If all data members can vary independently of each other, no invariant is possible.

Re-defining ‘invariance’

An invariant is a **logical condition** for the members of an object that can be expected to be true from the time the constructor exits to the time the destructor is called. These logical conditions fall into three categories:

- A relationship between at least two data members such that all members cannot vary independently.
- A guarantee that any function with access to a resource may be called at any time, i.e. implementing RAII.
- A property of a data member that must always be true.

Re-defining ‘invariance’

An invariant is a logical condition for the members of an object that can be expected to be true from the time the constructor exits to the time the destructor is called. These logical conditions fall into three categories:

- A relationship between at least two data members such that all members cannot vary independently.
- A guarantee that any function with access to a resource may be called at any time, i.e. implementing RAII.
- A property of a data member that must always be true.

$$V = I R$$

Ohm's Law In Code

```
class SimpleCiruit {
public:
  void setResistanceFixedVoltage(Resistance resistance) {
    m_resistance = resistance;
    m_amperage = m_volts / m_resistance;
  }
  void setResistanceFixedCurrent(Resistance resistance) {
    m_resistance = resistance;
    m_volts = m_amperage * m_resistance;
  }
  void setVoltsFixedCurrent(Voltage volts) {
    m_volts = volts;
    m_resistance = m_volts / m_amperage;
  }
  // ...
private:
  Voltage m_volts; Current m_amperage; Resistance m_resistance;
};
```

Not all variables have to impact all others

```
class Light {
public:
  void setBrightness(uint8_t brightness) {
    if (m_isLightOn) {
      m_brightness = brightness;
    }
  }
  void flipLightStatus() { m_isLightOn = !m_isLightOn; }

private:
  bool m_isLightOn;
  uint8_t m_brightness;
};
```

Re-defining ‘invariance’

An invariant is a logical condition for the members of an object that can be expected to be true from the time the constructor exits to the time the destructor is called. These logical conditions fall into three categories:

- A relationship between at least two data members such that all members cannot vary independently.
- A guarantee that any function with access to a resource may be called at any time, i.e. implementing RAII.
- A property of a data member that must always be true.

RAII definition

cppreference.com:

“Resource Acquisition Is Initialization, or RAII, is a C++ programming technique that binds the life cycle of a resource that must be acquired before use (allocated heap memory, thread of execution, open socket, open file, locked mutex, disk space, database connection—anything that exists in limited supply) to the lifetime of an object.

RAII guarantees that the resource is available to any function that may access the object (**resource availability is a class invariant**, eliminating redundant runtime tests). It also guarantees that all resources are released when the lifetime of their controlling object ends, in reverse order of acquisition.”

RAII example

```
class FileHandler {
public:
    FileHandler(const char* filename) {
        //throws an exception on failure
        m_file.open(filename);
    }

    ~FileHandler() {
        m_file.close();
    }

    std::string readLine() {
        return m_file.readLine();
    }

private:
    File m_file;
};
```

CADRe >> RAII

Constructor Acquires, Destructor Releases

Re-defining ‘invariance’

An invariant is a logical condition for the members of an object that can be expected to be true from the time the constructor exits to the time the destructor is called. These logical conditions fall into three categories:

- A relationship between at least two data members such that all members cannot vary independently.
- A guarantee that any function with access to a resource may be called at any time, i.e. implementing RAII.
- A property of a data member that must always be true.

Property of a Data Member That Must Be True

- Similar to the first rule.
- Usually a comparison to a constant or literal.
- Often in non-trivial getters and setters.
- Could be an algorithm that enforces the invariant.

Data member property - const comparison

```cpp
class EvenNumber {
public:
  EvenNumber(int i) { setNumber(i); }
  void setNumber(int i) {
    if ((i % 2) != 0) {
      throw std::runtime_error("Invalid input");
    } else {
      m_i = i;
    }
  }
private:
  int m_i;
};
```

Data member property - algorithm

```
class UnitCircleAngle {
public:
    void setAngleFromDegrees(float degrees) {
        auto boundedAngleDegrees{std::fmodf(degrees, 360)};
        m_angle = boundedAngleDegrees * M_PI / 180;
    }
private:
    float m_angle;
}
```

Review: Use a class when your data contains an invariant

- A relationship between at least two data members such that all members cannot vary independently.
- A guarantee that any function with access to a resource may be called at any time, i.e. RAII.
- A property of a data member that must always be true.

Let’s try this out.

Date data

```
int day;
int month;
int year;
```

Date class declaration

```
class Date {
public:
    Date(int day, int month, int year);
    void setDay(int day);
private:
    int m_day, m_month, m_year;
};
```

```cpp
class Date {
public:
    Date(int day, int month, int year) {
        if ( DateHelpers::isValidYear(year) &&
             DateHelpers::isValidMonth(month) &&
             DateHelpers::isValidDay(day,
                                     m_month,
                                     m_year)) {
            m_year = year;
            m_month = month;
            m_day = day;
        } else {
            throw std::exception();
        }
    }

    void setDay(int day) {
        if (DateHelpers::isValidDay(day,
                                    m_month,
                                    m_year)) {
            m_day = day;
        } else {
            throw std::exception();
        }
    }
private:
    int m_day, m_month, m_year;
};
```

```cpp
namespace DateHelpers {
    bool isValidYear(int year) {
        return (year != 0);
    }
    bool isValidMonth(int month) {
        return ((month > 0) && (month < 13));
    }
    bool isLeapYear(int year) {
        return false; // ...implementation
    }
    int numDaysInMonth(int month, int year) {
        int numDays;
        auto leapYear{isLeapYear(year)};
        //... implementation
        return numDays;
    }
    bool isValidDay(int day, int month, int year) {
        return ((day > 0) &&
                (day <= numDaysInMonth(month, year)));
    }
}
```

What if some of my data doesn’t have an invariant?

Data set with some invariance

```
//Data needed for entity
int idNumber;
bool isDataReceived;
std::chrono::milliseconds timeDataReceived;
bool isStatusHealthy;
```

Time received is updated whenever data is received. It shouldn't vary independent of isDataReceived

Encapsulate invariance in a class

```
using std::chrono::milliseconds;
class DataReceivedStatus {
public:
  DataReceivedStatus(std::function<milliseconds()> getCurrentTime) :
    m_getCurrentTime{getCurrentTime}

  void setDataReceived() {
    isDataReceived = true;
    timeDataReceived = m_getCurrentTime();
  }
private:
  std::function<milliseconds()> m_getCurrentTime;
  bool isDataReceived{false};
  milliseconds timeDataReceived{0};
};
```

Use the class in a struct

```
using std::chrono::milliseconds;
class DataReceivedStatus {
public:
  DataReceivedStatus(std::function<milliseconds()> getCurrentTime) :
    m_getCurrentTime{getCurrentTime}

  void setDataReceived() {
    isDataReceived = true;
    timeDataReceived = m_getCurrentTime();
  }
private:
  std::function<milliseconds()> m_getCurrentTime;
  bool isDataReceived{false};
  milliseconds timeDataReceived{0};
};
```

```
struct MyData {
  int idNumber;
  DataReceivedStatus dataRecStatus;
  bool isStatusHealthy;
}
```

Another option without using a class

```
//struct with class
struct MyData {
  int idNumber;
  DataReceivedStatus dataRecStatus;
  bool isStatusHealthy;
}
```

```
//struct without class
struct MyData {
  int idNumber;
  std::optional<milliseconds> timeDataReceived;
  bool isStatusHealthy;
}
```

Remember:

invariant = logical assertion

invariant ≠ immutable

Why so much emphasis on free functions?

Resources about free functions

CppCon 2017: Klaus Iglberger “Free Your Functions!”

- Discusses how free functions improve encapsulation, abstraction/polymorphism, cohesion, flexibility/extensibility, reuse/generality, testability, and performance.

Scott Meyers (2000), "How Non-Member Functions Improve Encapsulation"

- Flips some of the ideas many have against free functions.

Herb Sutter (early 2000s), "Monoliths 'Unstrung'"

- Rewrote C++98 std::string class from 103 member functions into 32 members functions and 71 free functions.

Scott Meyers (2005), "Effective C++ (3rd ed.)", Item 23, p.98.

- "Prefer non-member non-friend functions to member functions."

STRETCH BREAK

Invariance exception: component boundaries

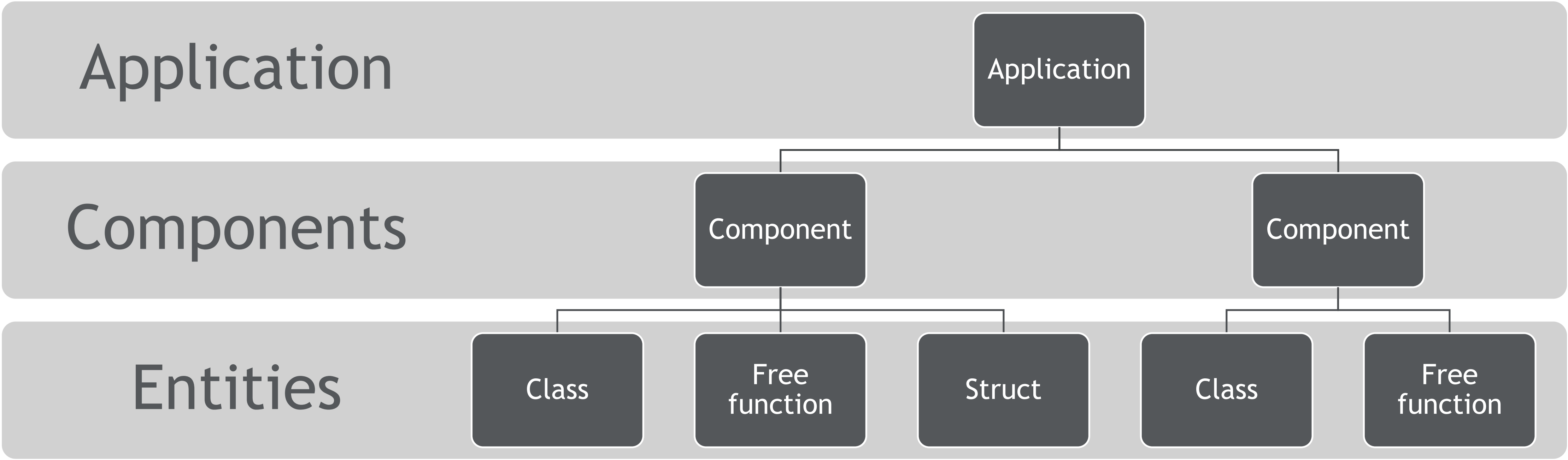
What is Decision-Making Isolation?

Separating decision-making into pure free functions or small classes without external dependencies.

Defaulting to using free functions and publicly-available data, such as structs or variables, instead of classes. Only use a class if the data has invariance or is part of a component boundary.

Unit testing all decision-making code thoroughly.

Entities are grouped into components



Guidance on designing components is out of scope

(See *Clean Architecture* or *Domain Driven Design* for more information)

QUESTION: What are the reasons you would choose to use a class if *not* for an invariant?

What makes a boundary that justifies a class?

- API that encapsulates a component.
- Dependency Inversion Principle.
- Run-time polymorphism.

In DMI, these classes are typically IO.

DMI, SRP, and OCP

SO definitions

SRP

“There should never be more than one reason for a class to change.”

“A module should be responsible to one, and only one, actor.” (From *Clean Architecture*, 2017)

OCP

“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.” (From *Clean Code*, 2008)

DMI and SRP

- Decision-making free functions do one thing.
- Classes that encapsulate an invariance should do one thing.
- Wiring - consider if more than one actor can cause change.

DMI and OCP

- Decision-making is easy to extend.
- Decision-making only changes because the decision changes.
- Invariance keeps classes focused.
- IO can change without impacting decision-making.
- Component boundaries provide a seam for polymorphism.

Unit Testing in Decision-Making Isolation

What is Decision-Making Isolation?

Separating decision-making into pure free functions or small classes without external dependencies.

Defaulting to using free functions and publicly-available data, such as structs or variables, instead of classes. Only use a class if the data has invariance or is part of a component boundary.

Unit testing all decision-making code thoroughly.

Remember the revised focus

To be skilled at utilizing unit testing, we need to focus on getting better at **choosing what code to test** and **designing code that is testable** rather than getting better at *writing more complex tests*.

The goal of testing is to build *confidence* that our software works.

Decision-making example - enum

```
EPizzaType determinePizzaToOrder(bool isWifePresent,
                                 bool isKid1Present,
                                 bool isKid2Present) {
    if (isKid1Present) {
        return EPizzaType::Cheese;
    } else if (isWifePresent && isKid2Present) {
        return EPizzaType::Sausage;
    } else if (isWifePresent) {
        return EPizzaType::Hawaiian;
    } else if (isKid2Present) {
        return EPizzaType::Pepperoni;
    } else {
        return EPizzaType::Veggie;
    }
}
```

Decision-making example - enum

```cpp
EPizzaType determinePizzaToOrder(bool isWifePresent,
                                 bool isKid1Present,
                                 bool isKid2Present) {
    if (isKid1Present) {
        return EPizzaType::Cheese;
    } else if (isWifePresent && isKid2Present) {
        return EPizzaType::Sausage;
    } else if (isWifePresent) {
        return EPizzaType::Hawaiian;
    } else if (isKid2Present) {
        return EPizzaType::Pepperoni;
    } else {
        return EPizzaType::Veggie;
    }
}
```

```cpp
TEST(DeterminePizzaOrder, Cheese_Whenever_Kid1_Present) {
  //Arrange
  constexpr bool isKid1Present{true};
  //Act and Assert
  EXPECT_EQ(EPizzaType::Cheese, determinePizzaToOrder(true,  isKid1Present, true));
  EXPECT_EQ(EPizzaType::Cheese, determinePizzaToOrder(true,  isKid1Present, false));
  EXPECT_EQ(EPizzaType::Cheese, determinePizzaToOrder(false, isKid1Present, true));
  EXPECT_EQ(EPizzaType::Cheese, determinePizzaToOrder(false, isKid1Present, false));
}

TEST(DeterminePizzaOrder, Sausage_When_Only_Wife_Kid2_Present) {
  //Arrange
  constexpr bool isWifePresent{true};
  constexpr bool isKid1Present{false};
  constexpr bool isKid2Present{true};

  //Act and Assert
  EXPECT_EQ(EPizzaType::Cheese, determinePizzaToOrder(isWifePresent,
                                                      isKid1Present,
                                                      isKid2Present));
}
```

Decision-making example - bool

```
bool shouldWeGoOutToEat(int energyLevel,
                        unsigned int numKidsPresent,
                        EDayOfWeek day) {
    if (numKidsPresent == 0) {
        return true;
    } else if ( (energyLevel > MIN_PATIENCE_ENERGY_THRESHOLD) &&
                (isWeekend(day)) ) {
        return true;
    } else {
        return false
    }
}
```

Decision-making example - algorithm

```
int calculateEnergyLevel(float rollingAvgHoursSleep,
                         unsigned int numCupsCaffeine) {
    auto baselineEnergyLevel{0.0863 * std::pow(rollingAvgHoursSleep, 3.352)};
    auto caffeineMultiplier{1 + 0.3 * (float)numCupsCaffeine};
    return  (int)(baselineEnergyLevel * caffeineMultiplier);
}
```

Coding example - before and after

```
class RequestHandler {
private:
    Outputter* m_outputter;
public:
    RequestHandler (Outputter* outputter) :
        m_outputter(outputter)
    {}

   void processRequest(EState state){
        switch (state){
            case EState::State1:
                m_outputter->send((int)EData::Data1);
                break;
            case EState::State2:
                m_outputter->send(2);
                break;
        }
    }
};
```

```
class RequestHandler {
private:
    Outputter* m_outputter;
public:
    RequestHandler (Outputter* outputter) :
        m_outputter(outputter) {}

    void processRequest_wiring(EState state) {
        auto data{RequestHandlerHelpers::decide(state)
        m_outputter->send(data);
    }};

namespace RequestHandlerHelpers {
int decide(EState state) {
    int ret;
    switch (state) {
        case EState::State1:
            ret = (int)EData::Data1;
            break;
        case EState::State2:
            ret = 2;
            break;
    }
        return ret;
}}
```

Testing the implementation without DMI

```
class RequestHandler {
private:
    Outputter* m_outputter;
public:
    RequestHandler (Outputter* outputter) :
        m_outputter(outputter)
    {}

   void processRequest(EState state){
        switch (state){
            case EState::State1:
                m_outputter->send((int)EData::Data1);
                break;
            case EState::State2:
                m_outputter->send(2);
                break;
        }
    }
};
```

```
class SpyOutputter : public Outputter {
public:
  void send(int a) {
    m_lastDataSent = a;
  }
private:
  int m_lastDataSent;
};

TEST(RequestHandlerTests, GivenState1_Return1) {
  //Arrange
  SpyOutputter outputter;
  RequestHandler uut(&outputter);
  constexpr int expectedSentData{1};
  //Act
  uut.processRequest(EState::State1);
  //Assert
  EXPECT_EQ(outputter.m_lastDataSent, expectedSentData);
}
```

Coding example – testing with DMI

```
class RequestHandler {
private:
    Outputter* m_outputter;
public:
    RequestHandler (Outputter* outputter) :
        m_outputter(outputter) {}

    void processRequest_wiring(EState state) {
        auto data{RequestHandlerHelpers::decide(state)};
        m_outputter->send(data);
    }};

namespace RequestHandlerHelpers {
 int decide(EState state) {
     int ret;
     switch (state) {
         case EState::State1:
             ret = (int)EData::Data1;
             break;
         case EState::State2:
             ret = 2;
             break;
     }
         return ret;
 }}
```

```
TEST(DecideTests, GivenState1_Return1) {
  //Arrange
  constexpr int expectedDecision{1};
  //Act and Assert
  EXPECT_EQ(RequestHandlerHelpers::decide(EState::State1),
            expectedDecision);
}
```

Coding example - test comparison

```
class SpyOutputter : public Outputter {
public:
  void send(int a) {
    m_lastDataSent = a;
  }
private:
  int m_lastDataSent;
  EData m_dataTypeSent;
};
```

```
TEST(RequestHandlerTests, GivenState1_Return1) {
   //Arrange
   SpyOutputter outputter;
   RequestHandler uut(&outputter);
   constexpr int expectedSentData{1};
   //Act
   uut.processRequest(EState::State1);
   //Assert
   EXPECT_EQ(outputter.m_lastDataSent, expectedSentData);
}
```

```
TEST(DecideTests, GivenState1_Return1) {
  //Arrange
  constexpr int expectedDecision{1};
  //Act and Assert
  EXPECT_EQ(RequestHandlerHelpers::decide(EState::State1),
            expectedDecision);
}
```

The goal of unit testing the decisions in DMI is to feel **confident** the software is **trying** to do **what we expect.**

What about testing the wiring?

- Automated tests that group entities together
- Unit test frameworks might still be used
- Focus testing on a few happy paths
    - High value

What about testing framework and drivers?

- Any automated tests are designed for testing this functionality (not including other logic).
- Tests should be run periodically on real hardware (not continuously by developers).
- Tests should be separately executable from unit tests.

TDD and DMI

Unexplored territory.

Key takeaways

Review: What is Decision-Making Isolation?

Separating decision-making into pure free functions or small classes without external dependencies.

Defaulting to using free functions and publicly-available data, such as structs or variables, instead of classes. Only use a class if the data has invariance or is part of a component boundary.

Unit testing all decision-making code thoroughly.

Isolating decisions is an iterative process.

What makes DMI unique?

“Extract Method”, *Test Driven Development by Example*, Kent Beck
“Separation of Concerns”, *C++ Software Design*, Klaus Iglberger

These suggest this as a solution to a problem

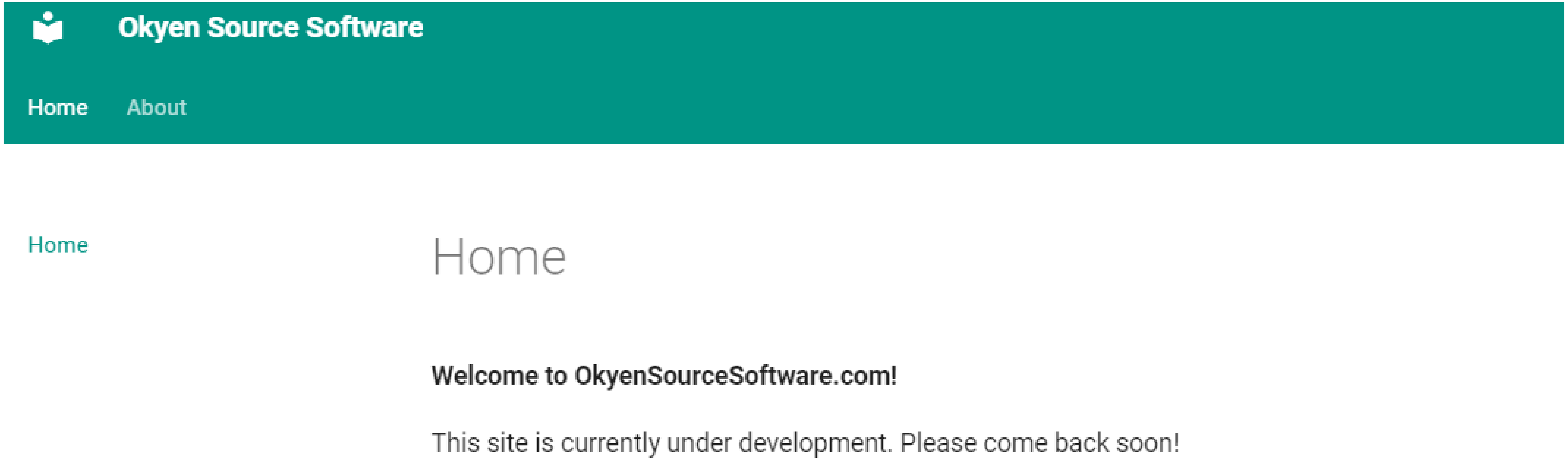
“Separate calculating from doing”, Value Oriented Programming Pt V, C++Now 2024, Tony Van Eerd

DMI: isolating decisions = default choice

Call to action

1. Use the C++ Core Guidelines
2. Focus unit testing on value
3. Consider invariance when designing classes
4. Use more free functions
5. Try Decision-Making Isolation

WIP: okyensourcesoftware.com



Special thanks

- My wife, Jess
- Matt St. Clair
- Joe Barnier
- Plexus and colleagues
- Klaus Iglberger

Questions?

Michael Okyen
mokyen@gmail.com
www.linkedin.com/in/michaelokyen
okyensourcesoftware.com