Engineering Practices for Maintainable Software

Matthew Supernaw
Scientific Programmer
National Oceanic and Atmospheric Administration
National Marine Fisheries Service
NSAP Modeling Team

1984

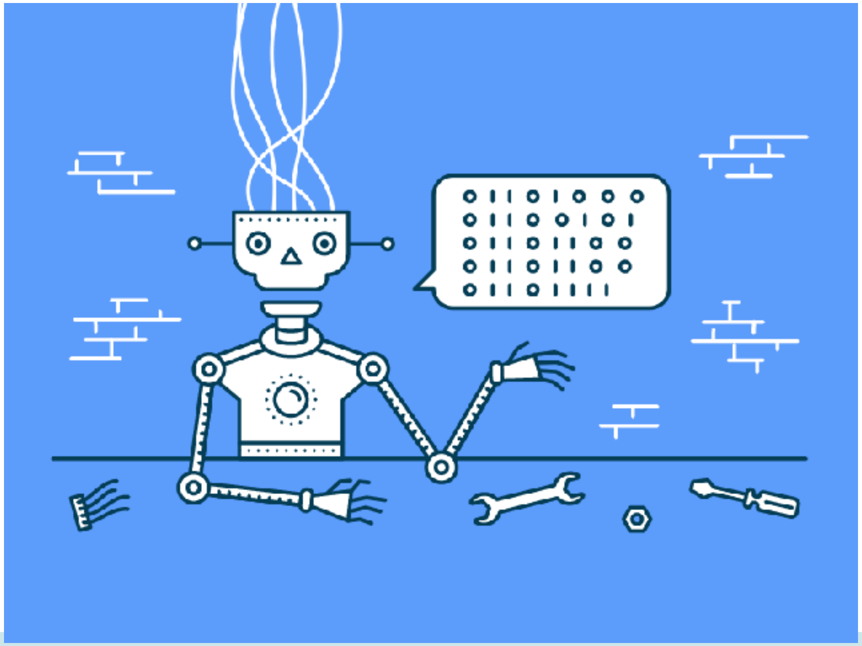
Overview

- **Software Engineering**
  - What is Software Engineering (SE)?
  - Analysis and Synthesis
  - SE in Practice
  - Core Principles
- **Programming Paradigms**
  - Structural
  - Object-oriented
- **Key Concepts**
  - Modularity
  - Extensibility
  - Scalability
  - Incremental Development
  - Maintainability
- **Tips for Better Code**
  - Use a Coding Convention
  - Write Useful Comments
  - Write Self-describing Code
  - Use an IDE
  - Refactor
  - Avoid Global Variables
  - Use Meaningful Names
  - Use Meaningful Structures
  - Use Version Control Software
  - Use documentation generators
  - Code for Efficiency
  - Profile often
- **Summary**

What is Software Engineering?

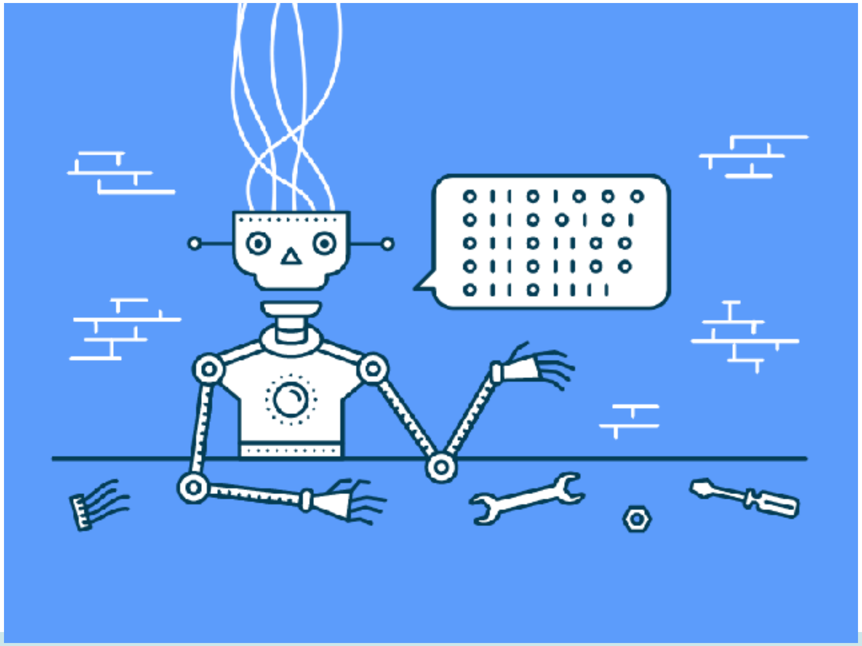
What is Software Engineering?

- The application of a systematic, disciplined, quantifiable approach to the development and maintenance of software.



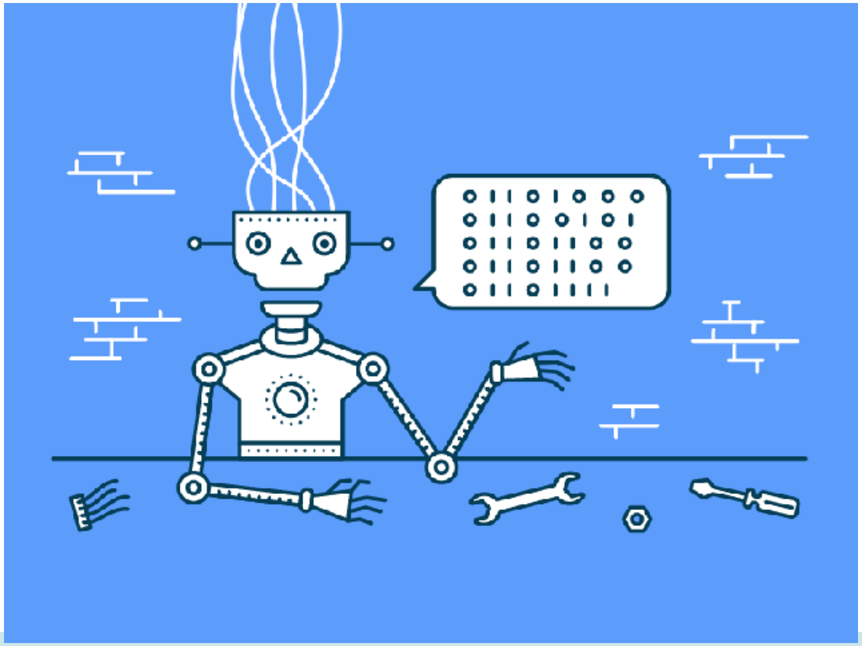
What is Software Engineering?

- The application of a **systematic**, disciplined, quantifiable approach to the development and maintenance of software.



What is Software Engineering?

- The application of a **systematic**, disciplined, quantifiable approach to the development and **maintenance** of software.

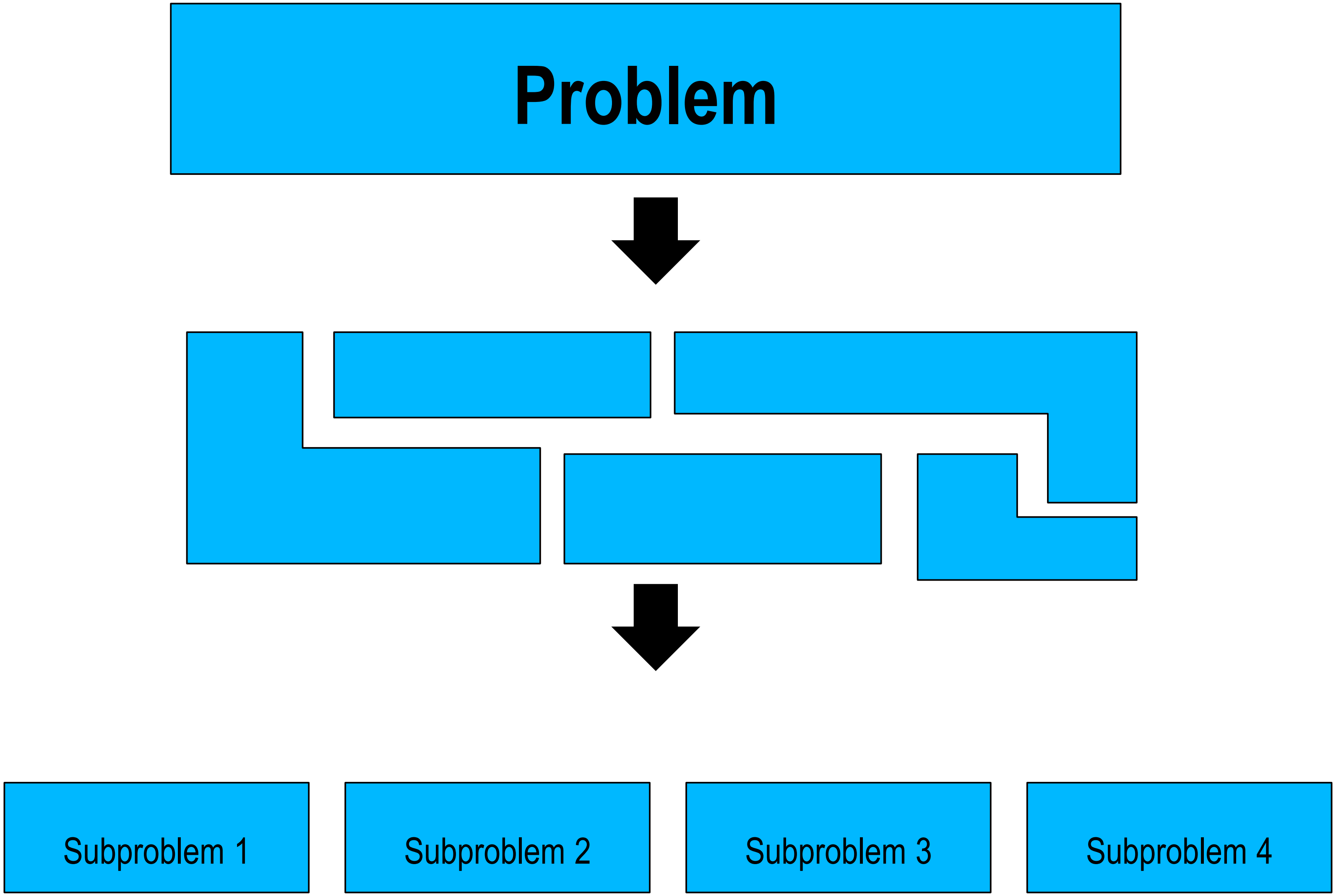


Analysis And Synthesis

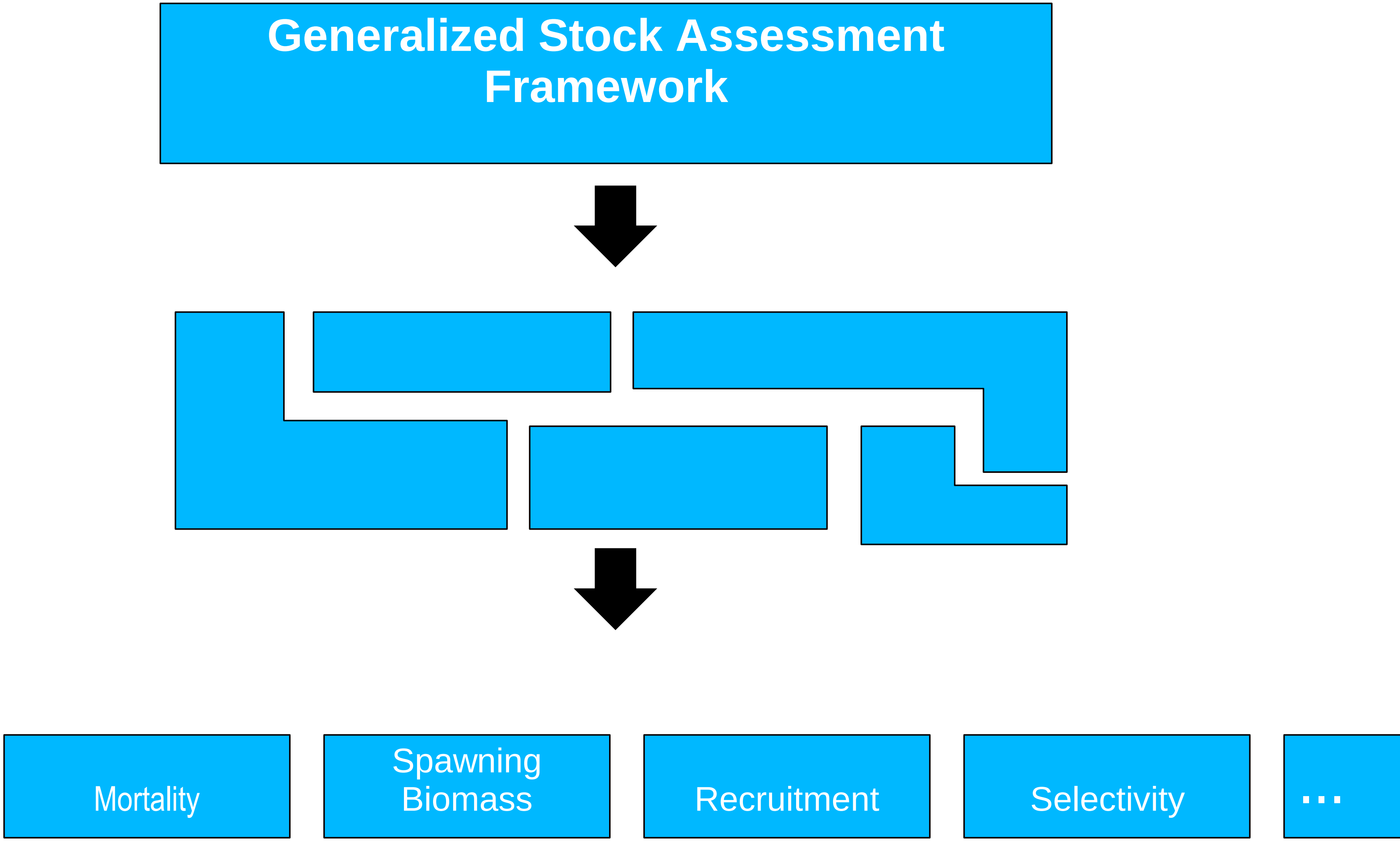
Analysis*: break down a larger problem into smaller understandable pieces (modules).*

Synthesis*: construct software from the smaller understood pieces.*

Analysis And Synthesis



Analysis And Synthesis



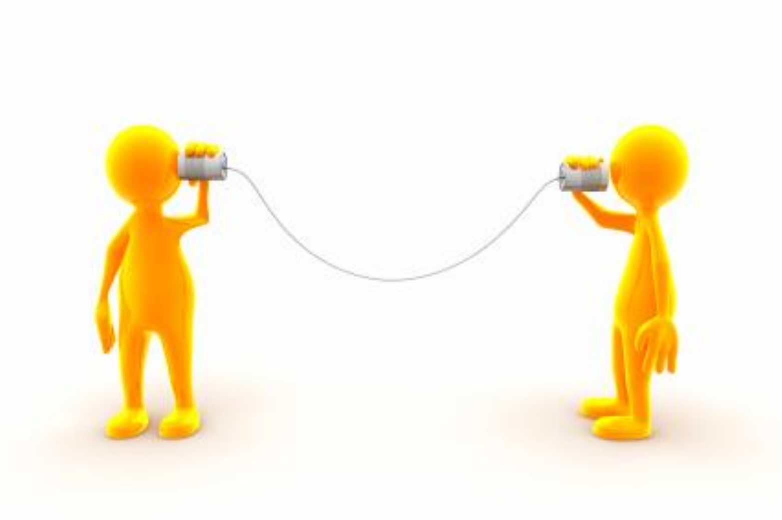
Software Engineering in Practice

Phases of A Software Engineering Project:

- *Communication*
- *Planning*
- *Modeling*
- *Development*
- *Testing*
- *Deployment*

Software Engineering in Practice

Communication:



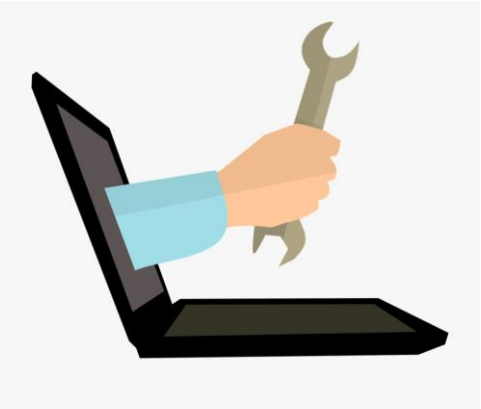
- *Meet with stakeholders.*
- *Gather requirements.*
- *Identify what exactly we are trying to solve.*
- *Document decisions.*

Software Engineering in Practice

Planning:

- *Planning is iterative.*
- *Gauge the project scope.*
- *Identify what resources are needed to solve the problem.*
- *Engage stakeholders in planning activity.*
- *Communicate the development plan.*

Software Engineering in Practice



Modeling and Design:

- *The <u>information domain</u> of the problem needs to be understood (information flows in and out of the system and subsystems).*

- *The <u>function</u> of the software should be defined.*

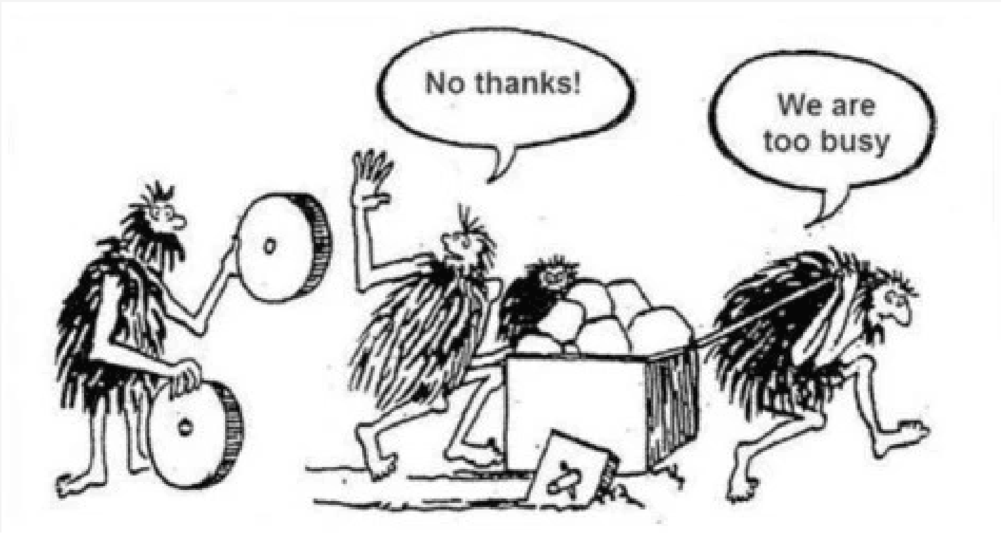
- *The <u>behavior</u> of the software should be represented.*

Software Engineering in Practice

Development:

- *Use structured or object-oriented programming (both emphasize modularity).*
- *Use appropriate data structures.*
- *Write self-describing code.*
- *Conduct regular code reviews.*
- *Unit test code.*
- *Refactor as necessary.*

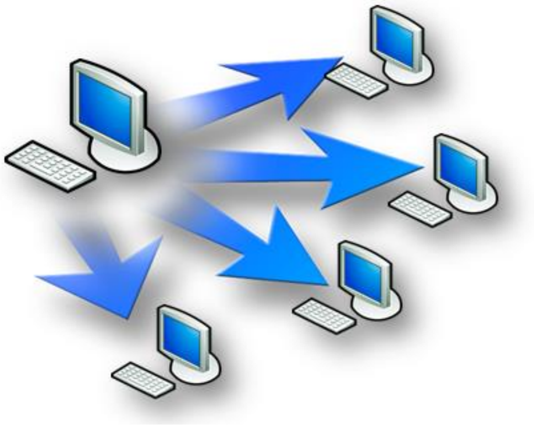
Software Engineering in Practice



Testing:

- *Identify areas that need improvement.*
- *All test should be traceable to software requirements.*
- *Tests should be planned before testing begins.*

Software Engineering in Practice



Deployment:

- *A complete package should be created.*
- *A support system should be established prior to release.*
- *Documentation should be provided to the end user.*
- *A maintenance plan should be established (extensions and updates).*

Core Principles of SE

Starting High Level

- ***Think about the problem before you start to develop the solution***

- ***Divide and Conquer***
    - *Break down the problem into smaller understandable modules.*
    - *This makes a large problem manageable.*

- ***KISS (Keep It Simple, Stupid!)***
    - *Don't add unnecessary complexity.*

- ***Keep Learning***
    - *Stay up to date on new technology.*
    - *Acknowledge your mistakes and learn from them.*

- ***Remember the purpose of the software***
    - *Keep the big picture in mind.*
    - *Don't add unnecessary functionality.*

- ***Remember you aren't necessarily the end user***
    - *Keep in mind the the end user won't necessarily be as familiar with the system as you.*

Core Principles of SE

The Process

- ***Have a Vision.***
    - *If you don't know exactly what to build, you wont build the right thing.*
    - *Ask questions and get clarity.*

- ***Be Systematic***
    - *Take a logical and thoughtful approach to the design.*
    - *Analyze and Synthesize.*

- ***Develop Iteratively***
    - *Supports modularity.*
    - *Compliments extensibility.*

- ***Make it work first, then optimize***
    - *Write it first, profile and optimize after.*

Core Principles of SE

Put thought into the Code

- ***YAGNI (You Ain't Gonna Need It!)***
  - *Don't add features that aren't required.*
  - *Don't add features that aren't required.*
  - *Don't add features that aren't required*
- ***DRY (Don't Repeat Yourself)***
  - *Reuse code whenever possible.*
- ***Don't re-invent the wheel***
  - *Use existing solutions if the code isn't related to the fundamentals of your application.*
  - *Just be aware of deep dependency and the issues that may arise.*
- ***Debugging is harder than writing code***
  - *Write readable code rather than compact code.*
  - *It's likely that someone else will have to work on your code later.*

Programming Paradigms

Programming Paradigms : *Structured Programming*

What is structured programing?

Programming Paradigms: *Structured Programming*

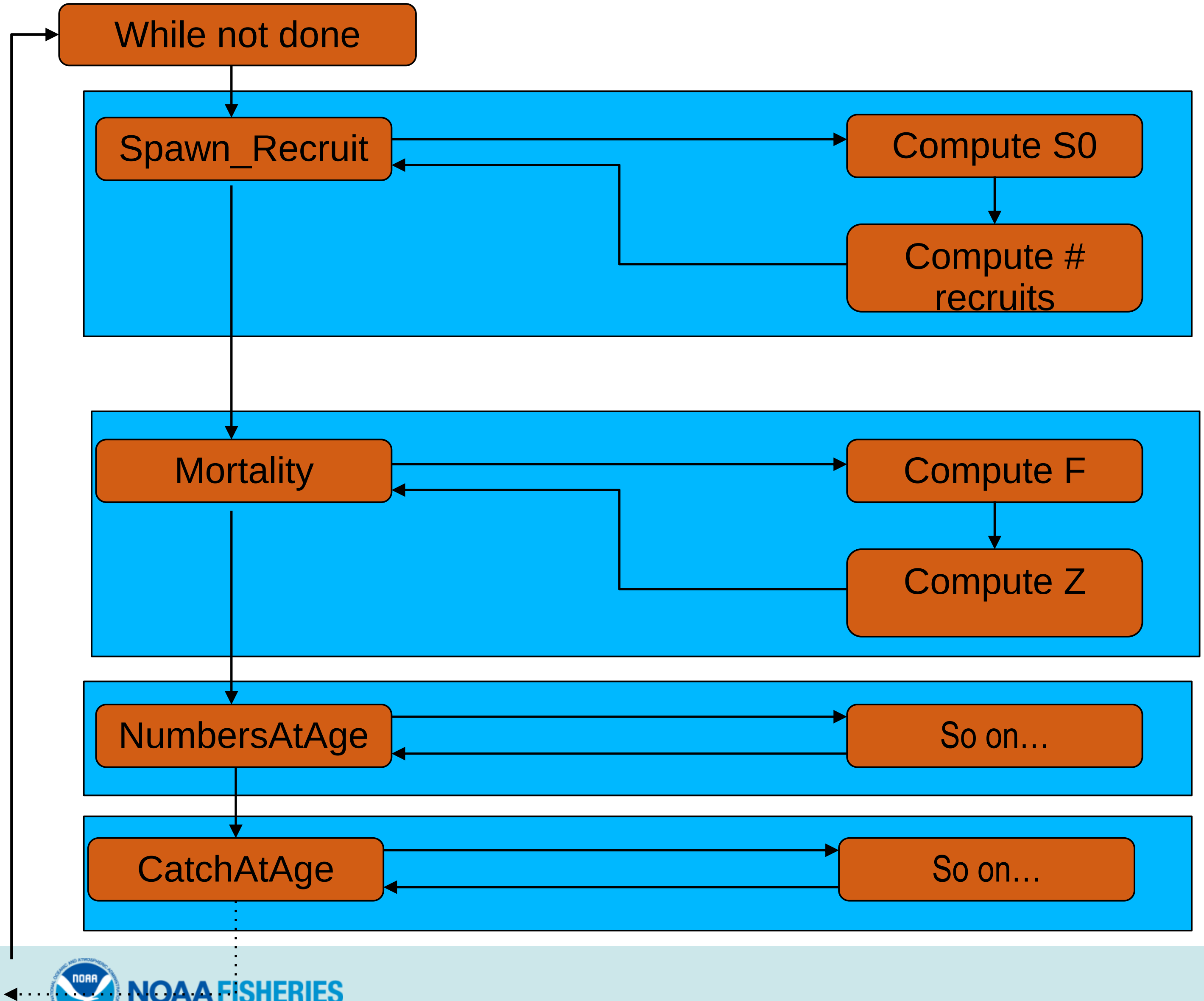
What is structured programing?

In structured programing, the program is divided into small modules so it's easier to understand

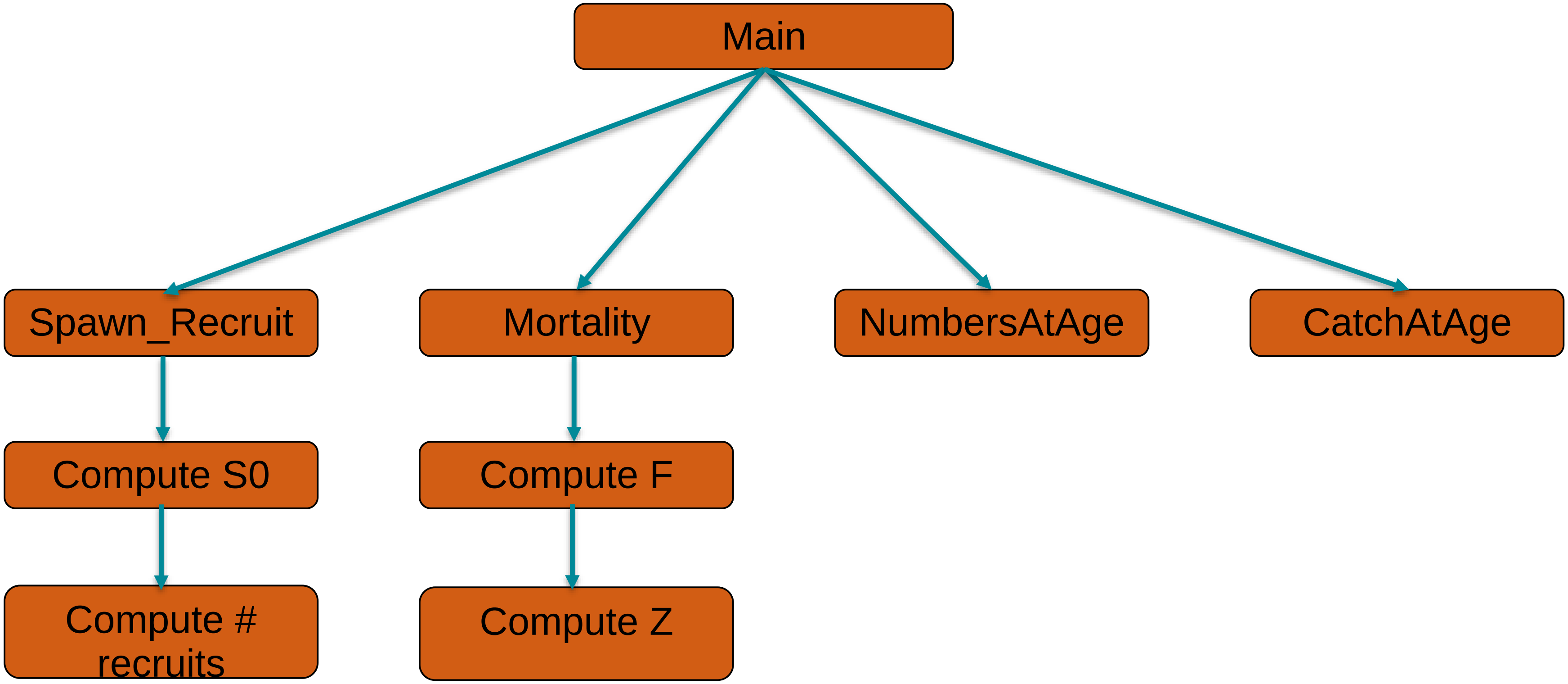
Programming Paradigms: *Structured Programming*

- *A* logical programming method that is considered a precursor to object-oriented programming (OOP).
- Facilitates program understanding and modification
- Has a top-down design approach
- A system is divided into compositional subsystems

Programming Paradigms: *Structured Programming*



Programming Paradigms: *Structured Programming*



Programming Paradigms: *Object-oriented Programming*

What is object oriented programing?

Programming Paradigms: *Object-oriented Programming*

What is object oriented programing?

Object-oriented programming (OOP) is a software programming model constructed around objects. This model compartmentalizes data into objects (data fields) and describes object contents and behavior through the declaration of classes (methods).

Programming Paradigms: *Object-oriented Programming*

What is an Object?

- *Software representation of a real-world object*
- *Just as with real-world objects, software objects have **state** and **behavior***
  - For example, Dogs have ***state*** (name, color, breed) and **behavior** (barking, fetching, drooling)

Programming Paradigms: *Object-oriented Programming*

Key concepts of OOP

- **Encapsulation**: This makes the program structure easier to manage because each object's implementation and state are hidden behind well-defined boundaries
- **Polymorphism**: This means abstract entities are implemented in multiple ways
- **Inheritance**: This refers to the hierarchical arrangement of implementation fragments (reusability).

Programming Paradigms: *Object-oriented Programming*

Encapsulation:

Refers to an object's ability to hide data and behavior that are not necessary to its user. Encapsulation allows a group of members and methods to be represented as a single unit.

Benefits:

- *Protection of data from accidental corruption*
- *Flexibility and extensibility of the code*
- *Reduction in complexity*
- *Lower coupling between code fragments and hence improvement in code maintainability*

Programming Paradigms: *Object-oriented Programming*

Polymorphism:

The dictionary definition of *polymorphism* refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming.

Programming Paradigms: *Object-oriented Programming*

Polymorphism:

```
class SelectivityBase {
public:
   virtual double Evaluate(double age) = 0;
};

class Logistic : public SelectivityBase {
   double a50;
   double s;
public:

   virtual double Evaluate(double age) {
      return 1.0 / (1.0 + exp(-1.0 * (age - a50) / s));
   }
};

class DoubleLogistic : public SelectivityBase {
   double alpha_asc;
   double beta_asc;
   double alpha_desc;
   double beta_desc;
public:

   virtual double Evaluate(double age) {
      return (1.0 /(1.0 + exp(-beta_asc * (age - alpha_a
            (1.0 - (1.0 / (1.0 + exp(-beta_desc * (age - a
   }
};
```

NOAA FISHERIES

Programming Paradigms: *Object-oriented Programming*

Polymorphism:

```
class SelectivityBase {
public:
   virtual double Evaluate(double age) = 0;
};

class Logistic : public SelectivityBase {
   double a50;
   double s;
public:

   virtual double Evaluate(double age) {
      return 1.0 / (1.0 + exp(-1.0 * (age - a50) / s));
   }
};

class DoubleLogistic : public SelectivityBase {
   double alpha_asc;
   double beta_asc;
   double alpha_desc;
   double beta_desc;
public:

   virtual double Evaluate(double age) {
      return (1.0 /(1.0 + exp(-beta_asc * (age - alpha_a
            (1.0 - (1.0 / (1.0 + exp(-beta_desc * (age - a
   }
};
```

Same Signature,
Different Logic

Programming Paradigms: *Object-oriented Programming*

Polymorphism:

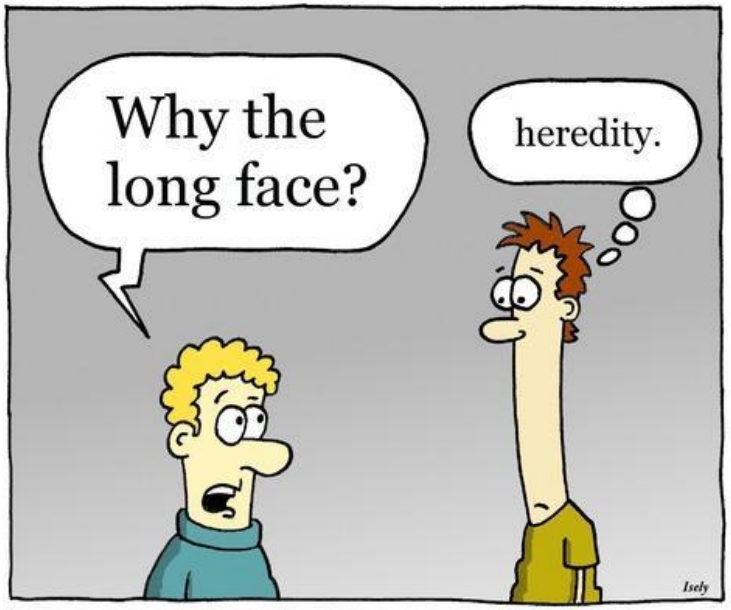
```
class SelectivityBase {
public:
   virtual double Evaluate(double age) = 0;
};

class Logistic : public SelectivityBase {
   double a50;
   double s;
public:

   virtual double Evaluate(double age) {
      return 1.0 / (1.0 + exp(-1.0 * (age - a50) / s));
   }
};

class DoubleLogistic : public SelectivityBase {
   double alpha_asc;
   double beta_asc;
   double alpha_desc;
   double beta_desc;
public:

   virtual double Evaluate(double age) {
      return (1.0 /(1.0 + exp(-beta_asc * (age - alpha_a
            (1.0 - (1.0 / (1.0 + exp(-beta_desc * (age - a
   }
};
```

Encapsulation

Programming Paradigms: *Object-oriented Programming*

Inheritance:

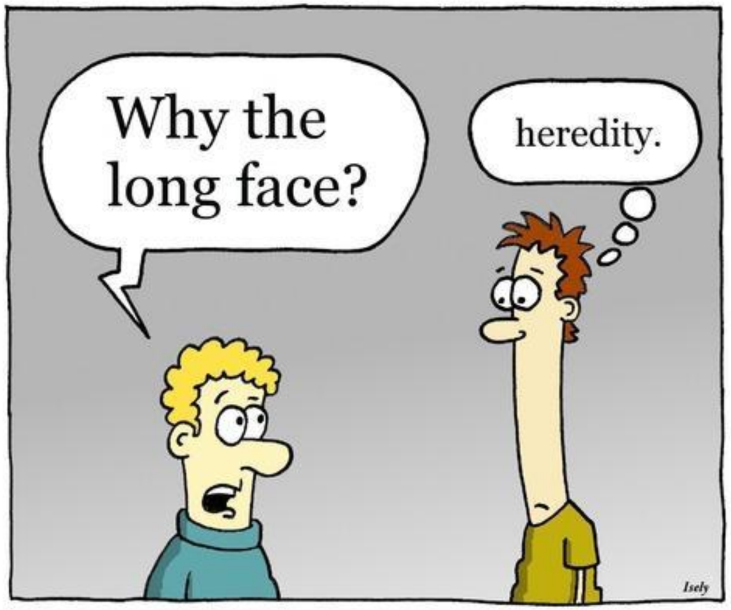
Inheritance allows the user to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.



Programming Paradigms: *Object-oriented Programming*

Inheritance:

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es)



NOAA FISHERIES

Programming Paradigms: *Object-oriented Programming*

Inheritance:

```cpp
class Shape {
public:

    void SetHeight(int height) {
        this->height = height;
    }

    void SetWidth(int width) {
        this->width = width;
    }
protected:
    int height;
    int width;

};

class Rectangle : public Shape{
public:
    int area(){
        return this->height*this->width;
    }
};
```

```cpp
int main(int argc, char** argv) {

    Rectangle r;
    r.SetHeight(10);
    r.SetWidth(2);
    std::cout<<r.area()<<"\n";

    return 0;
}
```

Output

20

Programming Paradigms: *Object-oriented Programming*

Inheritance:

```
class Shape {
public:

   void SetHeight(int height) {
      this->height = height;
   }

   void SetWidth(int width) {
      this->width = width;
   }
protected:
   int height;
   int width;

};

class Rectangle : public Shape{
public:
   int area(){
      return this->height*this->width;
   }
};
```

Derivation List

```
int main(int argc, char** argv) {

   Rectangle r;
   r.SetHeight(10);
   r.SetWidth(2);
   std::cout<<r.area()<<"\n";

   return 0;
}
```

Output

20

Programming Paradigms: *Object-oriented Programming*

Inheritance:

```
class Shape {
public:

    void SetHeight(int height) {
        this->height = height;
    }

    void SetWidth(int width) {
        this->width = width;
    }
protected:
    int height;
    int width;

};

class Rectangle : public Shape{
public:
    int area(){
        return this->height*this->width;
    }
};
```

Rectangle inherits methods from Shape

```
int main(int argc, char** argv) {

    Rectangle r;
    r.SetHeight(10);
    r.SetWidth(2);
    std::cout<<r.area()<<"\n";

    return 0;
}
```

Output

20

Key Concepts

Key Concepts

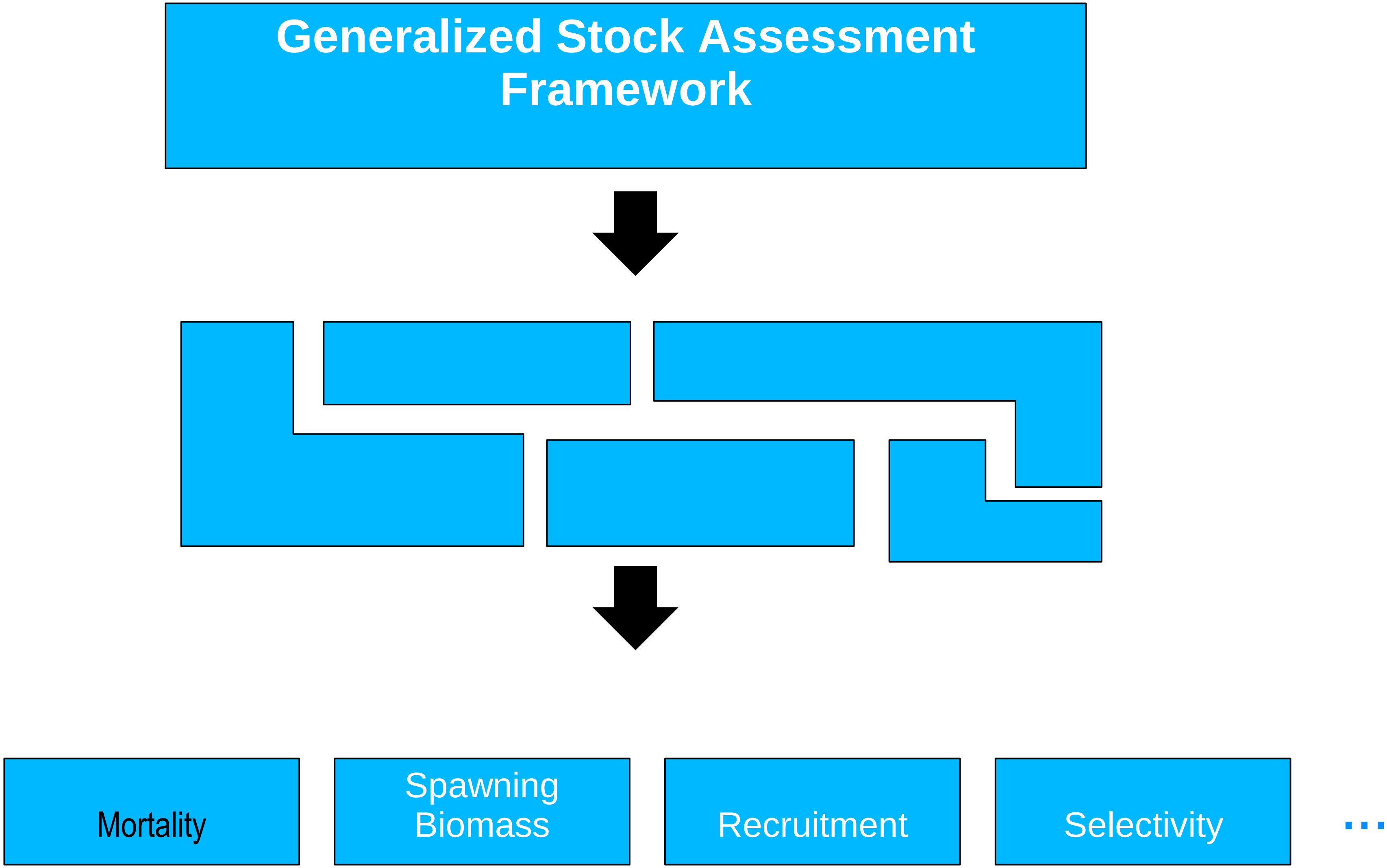
Key Concepts

- ***Modularity***
- ***Reusability***
- ***Extensibility***
- ***Extensible Design***
- ***Iterative Development***
- ***Scalability***
- ***Maintainability***

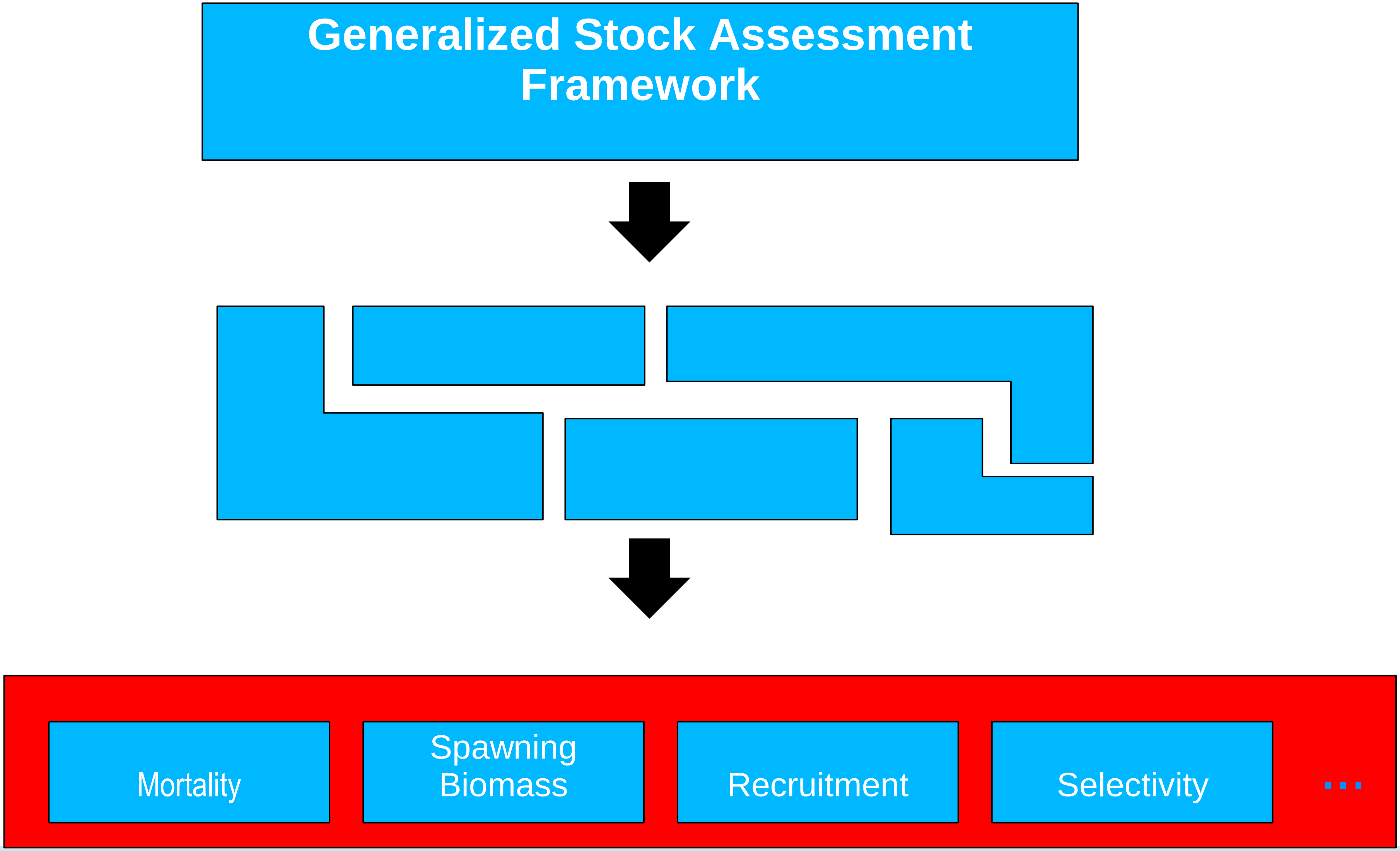
Key Concepts: *Modularity*

- *Well defined, independent components (functions or objects)*
- *Perform logically discrete functions*
- *Building blocks for a larger component(s)*
- *Can be implemented and tested in isolation before integration*
- *Accommodates division of work*
- *Improves maintenance*

Key Concepts: *Modularity*



Key Concepts: *Modularity*



Key Concepts: *Reusability*

*Reusability, a product of **modularity**, is the use of existing elements within the software development process. These elements are products and by-products of the software development life cycle and include code, test suites, designs and documentation.*

Key Concepts: *Extensibility*

Extensibility is the measure of a software components capacity to be appended with additional members or features. An application is considered extensible when its operations may be augmented with add-ons and plugins without the need for reengineering.

Key Concepts: *Extensible Design*

- *Extensible design is to accept that not all features can be designed in advance*
- *The system starts as basic framework that allows for changes (extensions)*
- *Extensible design allows small changes to be implemented upon request (Agile development)*
- *Extensibility imposes fewer dependencies during development*

Key Concepts: *Iterative Development*

Iterative development is a methodology that divides the project into smaller pieces (modules). The main concept of iterative development is to create small projects with well defined scope within a project. Iterative development naturally complements extensible design.

Key Concepts: *Scalability*

What is scalability?

Key Concepts: *Scalability*

What is scalability?

Software scalability refers to a systems ability to handle an increase in workload.

Key Concepts: *Scalability*

- *A system is said to be scalable when it does not require reengineering to handle an increase in workload.*
- *"Workload" may refer to required data storage, number of users, or anything that pushes the software past its original capacity.*
- *Designing software with scalability in mind saves time and money in the future.*

Key Concepts: *Scalability*

What's the problem with scaling failures?

Key Concepts: *Scalability*

What's the problem with scaling failures?

- *Increased workload becomes a barrier to productivity.*

Key Concepts: *Scalability*

What's the problem with scaling failures?

- *Increased workload becomes a barrier to productivity.*

- *"Fixes" add complexity.*

Key Concepts: *Scalability*

What's the problem with scaling failures?

- *Increased workload becomes a barrier to productivity.*
- *"Fixes" add complexity.*
- *Complexity increases cost and decreases effectiveness.*

Key Concepts: *Scalability*

What's the problem with scaling failures?

- *Increased workload becomes a barrier to productivity.*
- *"Fixes" add complexity.*
- *Complexity increases cost and decreases effectiveness.*
- *Users abandon the product.*

Key Concepts: *Scaling Failure Solutions*

Scaling Up

- *Refers to the idea of adding more advanced hardware to handle the increase in workload. For example, a faster CPU or more memory.*
- *Best performance solution, most costly.*

Key Concepts: *Scaling Failure Solutions*

Scaling Out

- *Refers to the idea of adding more hardware, not more advanced hardware.*
- *Much more widely used solution.*
- *Cost is lower because there isn't a need for more advanced hardware.*

Key Concepts: *Maintainability*

Maintenance *is the action of modifying a software product after initial release.*

Maintainability *is the ease with which a software product can be modified.*

Key Concepts: *Maintainability*

Categories:

- ***Corrective Maintenance*** *is a task performed to identify and fix failures (bugs) in the system.*
- ***Adaptive Maintenance*** *is the implementation of a changes as result to a change in the environment. (hardware or operating system).*
- ***Perfective Maintenance*** *is the extension and improvement of the software quality.*

Key Concepts: *Maintainability*

Software Quality Characteristics for Enhanced Maintainability:

- ***Flexibility:*** *The ease at which the software can be amended.*
- ***Reliability:*** *Performance should be reliable with minimal faults.*
- ***Portability:*** *The application should run on different platforms, Linux, Windows, Mac OS, etc.*
- ***Efficiency:*** *Practical and efficient use of system resources.*
- ***Testability:*** *Software should be tested easily and as a result users can easily check that the results are correct.*
- ***Understandability:*** *Software should be easy for users to understand.*
- ***Usability:*** *Usage is easy and comfortable.*

Tips For Better Code

Tips For Better Code: *Coding Convention*

Why Have a Coding Convention?

Tips For Better Code: *Coding Convention*

Why Have a Coding Convention?

- *80% of a product's lifetime cost goes to maintenance.*
- *Software is usually maintained by someone other than the original author.*
- *Code conventions improve readability.*

Tips For Better Code: *Coding Convention*

Common Coding Conventions

- *C*
    - [***SEI CERT***]
    - [***Bar Group***]
- *C++*
    - [***Google***]
    - [***SEI CERT C++***]
- *Java*
    - [***Java Code Convention***]
    - [***Software Monkey***]
- *R*
    - [***R-core***]

Tips For Better Code: *Useful Comments*

Write Useful Comments

You won't appreciate them until you've stopped working on a project for a while. Useful Comments make it easier for you and those after you who have to maintain your code.

- *Write meaningful, single line comments for easily understood components.*
- *Write full paragraphs for components that are not easily understood.*
- *For complex blocks of logic, describe what's going on in words before the logic appears.*

Write Self-Describing Code

Give Symbols Human readable names.

- *It makes source code easier to understand.*
- *It makes code easier to maintain and extend.*

Tips For Better Code: Self-Describing Code

Write Self-Describing Code

Give Symbols Human readable names.

- *It makes source code easier to understand.*
- *It makes code easier to maintain and extend.*

Example

```
double sel(double a) {
    return 1.0 / (1.0 + exp(-1.0 * (a - a50) / s));
}

double calculate_logistic_selectivity(double age) {
    return 1.0 / (1.0 + exp(-1.0 * (age - a50) / s));
}
```

Tips For Better Code: Use An IDE

Use an Integrated Development Environment

***Benefits**:*

- *Expands Coders Capabilities.*
- *Increased Functionality.*
- *Navigate to members by treating them as hyperlinks.*
- *Autocompletion when you can't remember the names of all members.*
- *Automatic code generation.*
- *Refactoring.*

Tips For Better Code: Use An IDE (Continued)

Use an Integrated Development Environment

***Benefits**:*

- *Warning-as-you-type.*
- *Automated Testing.*
- *Integrated Debugger.*
- *Profiling.*
- *Integrated Source Control.*
- *Auto Code Formatting.*
- *Auto Code Completion.*
- *Call Graph Generation.*

Tips For Better Code: Refactor

What is Refactoring?

Definition:

Refactoring consists of improving the internal structure of an existing program's source code, while preserving its external behavior.

***Benefits**:*

- *Refactoring improves objective attributes of code (length, duplication, coupling and cohesion, cyclomatic complexity) that correlate with ease of maintenance.*
- *Refactoring helps code understanding.*
- *Refactoring encourages each developer to think about and understand design decisions, in particular in the context of collective ownership / collective code ownership.*
- *Refactoring favors the emergence of reusable design elements (such as design patterns) and code modules.*

Tips For Better Code: Avoid Global Variables

Avoid global variables whenever possible!

- *A global variable is a variable defined in the 'main' program. Such variables are said to have 'global' scope.*
- *A local variable is a variable defined within a function. Such variables are said to have ‘local’ scope.*
- *They can be modified anywhere in the program, making it difficult to find the source of change.*
- *Functions can access global variables and modify them.*
- *They violate the concept of modular programming.*
- *It’s better practice to send a variable as a parameter to a function.*

Tips For Better Code: *Use Meaningful Names*

Use Meaningful Names:

- *Good code should be meaningful in terms of variable names, function/method names, and class names.*

- *Don't use names like "**fyr**" or "**lyr**" for your variables. It is not informative. "**first_year**" and "**last_year**" would be more meaningful.*

For example:

```
double calculate_logistic_selectivity(double age) {
    return 1.0 / (1.0 + exp(-1.0 * (age - a50) / s));
}
```

Tips For Better Code: *Use Meaningful Names*

Use Meaningful Names:

- *Good code should be meaningful in terms of variab function/method names, and class names.*

- *Don't use names like "**fyr**" or "**lyr**" for your variables. informative. "**first_year**" and "**last_year**" would be re meaningful.*

It's obvious what this function does and what it's parameter is!

For example:

```
double calculate_logistic_selectivity(double age) {
    return 1.0 / (1.0 + exp(-1.0 * (age - a50) / s));
}
```

Tips For Better Code: *Use Meaningful Names*

Use Meaningful Names:

- *Good code should be meaningful in terms of variab function/method names, and class names.*

It's obvious what this function does and what it's parameter is!

- *Don't use names like "**fyr**" or "**lyr**" for your variables. informative. "**first_year**" and "**last_year**" would be re meaningful.*

For example:

```
double calculate_logistic_selectivity(double age) {
    return 1.0 / (1.0 + exp(-1.0 * (age - a50) / s));
}
```

*The verb "**calculate**" makes it clear that the function is doing a calculation rather than a lookup. The "**logistic_selectivity**" makes it clear what the return value is.*

Tips For Better Code: *Use Meaningful Structures*

Use Meaningful Structures:

- *Use a naming convention when naming directories and files.*
- *Use simple directory structures.*
- *Keep the directory hierarchy as shallow as possible.*
- *Try to split up code by it's business logic (modules).*

Tips For Better Code: *Use Version Control System*

Version Control:

- *There are many varieties to choose from.*
- *Managing changes should be easy.*
- *Choose whatever version control software works best for the workflow of you and your team.*

Popular Choices:

- *Git*
- *Mercurial*
- *Apache Subversions*

Tips For Better Code: *Use Documentation Generators*

Code Documenters

- *For large projects with many classes and functions, it's convenient to automatically generate API documentation.*

- *Document generators are useful for keeping track of what's going on in the code.*

Useful Documentation Generators:

- *Doxygen*
- *JavaDocs*
- roxygen

Tips For Better Code: *Code For Efficiency*

Code Efficiency:

- *Code efficiency plays a vital role in applications in a high-execution-speed environment where performance and scalability are foremost.*
- *The goal of code efficiency is to reduce resource consumption and completion time as much as possible.*
- *Simply put, the more efficient the code, the lower the computational overhead!*

Tips For Better Code: *Code For Efficiency*

Example of Inefficient Code:

```
enum RecruitmentFunction {
    BEVERTONHOLT = 0,
    RICKER,
    DERISO,
    SHEPARD
};

double calculate_recruitment(double spawning_biomass, RecruitmentFunction recruitment_function) {

  if (recruitment_function == BEVERTONHOLT) {
    return beverton_holt(spawning_biomass);
  } else if (recruitment_function == RICKER) {
    return ricker(spawning_biomass);
  } else if (recruitment_function == DERISO) {
    return deriso(spawning_biomass);
  } else if (recruitment_function == SHEPARD) {
    return shepard(spawning_biomass);
  } else {
    std::cout << "Error: unknown recruitment function" << std::endl;
    return 0.0;
  }

}
```

Tips For Better Code: ***Code For Efficiency***

Example of Inefficient Code:

Case statements are expensive operations.

```
enum RecruitmentFunction {
    BEVERTONHOLT = 0,
    RICKER,
    DERISO,
    SHEPARD
};

double calculate_recruitment(double spawning_biomass, RecruitmentFunction recruitment_function) {

    if (recruitment_function == BEVERTONHOLT) {
        return beverton_holt(spawning_biomass);
    } else if (recruitment_function == RICKER) {
        return ricker(spawning_biomass);
    } else if (recruitment_function == DERISO) {
        return deriso(spawning_biomass);
    } else if (recruitment_function == SHEPARD) {
        return shepard(spawning_biomass);
    } else {
        std::cout << "Error: unknown recruitment function" << std::endl;
        return 0.0;
    }

}
```

Tips For Better Code: *Code For Efficiency*

Example of Efficient Code Using the Structured Programming Paradigm:

```
enum RecruitmentFunction {
    BEVERTONHOLT = 0,
    RICKER,
    DERISO,
    SHEPARD
};

typedef double(*recruitment_function_ptr)(double);

std::vector<recruitment_function_ptr> recruitment_functions
       = {&beverton_holt, &ricker, &deriso, &shepard};

double calculate_recruitment(double spawning_biomass, RecruitmentFunction recruitment_function) {

    return recruitment_functions[recruitment_function](spawning_biomass);

}
```

Tips For Better Code: ***Code For Efficiency***

Example of Efficient Code Using the Structured Programming Paradigm:

```
enum RecruitmentFunction {
   BEVERTONHOLT = 0,
   RICKER,
   DERISO,
   SHEPARD
};

typedef double(*recruitment_function_ptr)(double);

std::vector<recruitment_function_ptr> recruitment_functions
      = {&beverton_holt, &ricker, &deriso, &shepard};

double calculate_recruitment(double spawning_biomass, RecruitmentFunction recruitment_function) {

   return recruitment_functions[recruitment_function](spawning_biomass);

}
```

The correct function is called directly without the use of case statements.

Tips For Better Code: *Code For Efficiency*

Example of Efficient Code Using the Object-Oriented Programming Paradigm:

```
enum RecruitmentFunction {
   BEVERTONHOLT = 0,
   RICKER,
   DERISO,
   SHEPARD
};

  class recruitment_model_base {
  public:
     virtual double calculate_recruits(double spawning_biomass) = 0;
  };

  class beverton_holt : public recruitment_model_base {
  public:

     double calculate_recruits(double spawning_biomass) {
        ...
     }
  };
```

Tips For Better Code: *Code For Efficiency*

Example of Efficient Code Using the Object-Oriented Programming Paradigm (Continued):

```
class ricker : public recruitment_model_base {
public:

    double calculate_recruits(double spawning_biomass) {
        ...
    }
};

class deriso : public recruitment_model_base {
public:

    double calculate_recruits(double spawning_biomass) {
        ...
    }
};

class shepard : public recruitment_model_base {
public:

    double calculate_recruits(double spawning_biomass) {
        ...
    }
};
```

Tips For Better Code: *Code For Efficiency*

Example of Efficient Code Using the Object-Oriented Programming Paradigm** (Continued)**:

```
recruitment_model_base* recruitment_model;

void initialize_recruitment_model(RecruitmentFunction recruitment_function) {
   switch (recruitment_function) {
      case BEVERTONHOLT:
         recruitment_model = new beverton_holt();
         break;
      case RICKER:
         recruitment_model = new ricker();
         break;
      case DERISO:
         recruitment_model = new deriso();
         break;
      case SHEPARD:
         recruitment_model = new shepard();
         break;
      default:
         std::cout << "Error: unknown recruitment function, using beverton_holt" << std::endl;
         recruitment_model = new beverton_holt();
   }
}
```

Tips For Better Code: *Code For Efficiency*

Example of Efficient Code Using the Object-Oriented Programming Paradigm** (Continued)**:

```
recruitment_model_base* recruitment_model;

void initialize_recruitment_model(RecruitmentFunction recruitment_function) {
   switch (recruitment_function) {
      case BEVERTONHOLT:
         recruitment_model = new beverton_holt();
         break;
      case RICKER:
         recruitment_model = new ricker();
         break;
      case DERISO:
         recruitment_model = new deriso();
         break;
      case SHEPARD:
         recruitment_model = new shepard();
         break;
      default:
         std::cout << "Error: unknown recruitment function, using beverton_holt" << std::endl;
         recruitment_model = new beverton_holt();
   }
}
```

Recruitment model pointer

Tips For Better Code: *Code For Efficiency*

Example of Efficient Code Using the Object-Oriented Programming Paradigm (Continued):

```
recruitment_model_base* recruitment_model;

void initialize_recruitment_model(RecruitmentFunction recruitment_function) {
   switch (recruitment_function) {
      case BEVERTONHOLT:
         recruitment_model = new beverton_holt();
         break;
      case RICKER:
         recruitment_model = new ricker();
         break;
      case DERISO:
         recruitment_model = new deriso();
         break;
      case SHEPARD:
         recruitment_model = new shepard();
         break;
      default:
         std::cout << "Error: unknown recruitment function, using beverton_holt" << std::endl;
         recruitment_model = new beverton_holt();
   }
}
```

Recruitment model pointer.

Initialize the recruitment model. Called at start up.

Tips For Better Code: *Code For Efficiency*

Example of Efficient Code Using the Object-Oriented Programming Paradigm** (Continued)**:

The correct method is called without the use of case statements.

```
inline double calculate_recruitment(double spawning_biomass ) {
   return recruitment_model->calculate_recruits(spawning_biomass);
}
```

Tips For Better Code: *Use A* ***Profiler***

What is a Profiler?

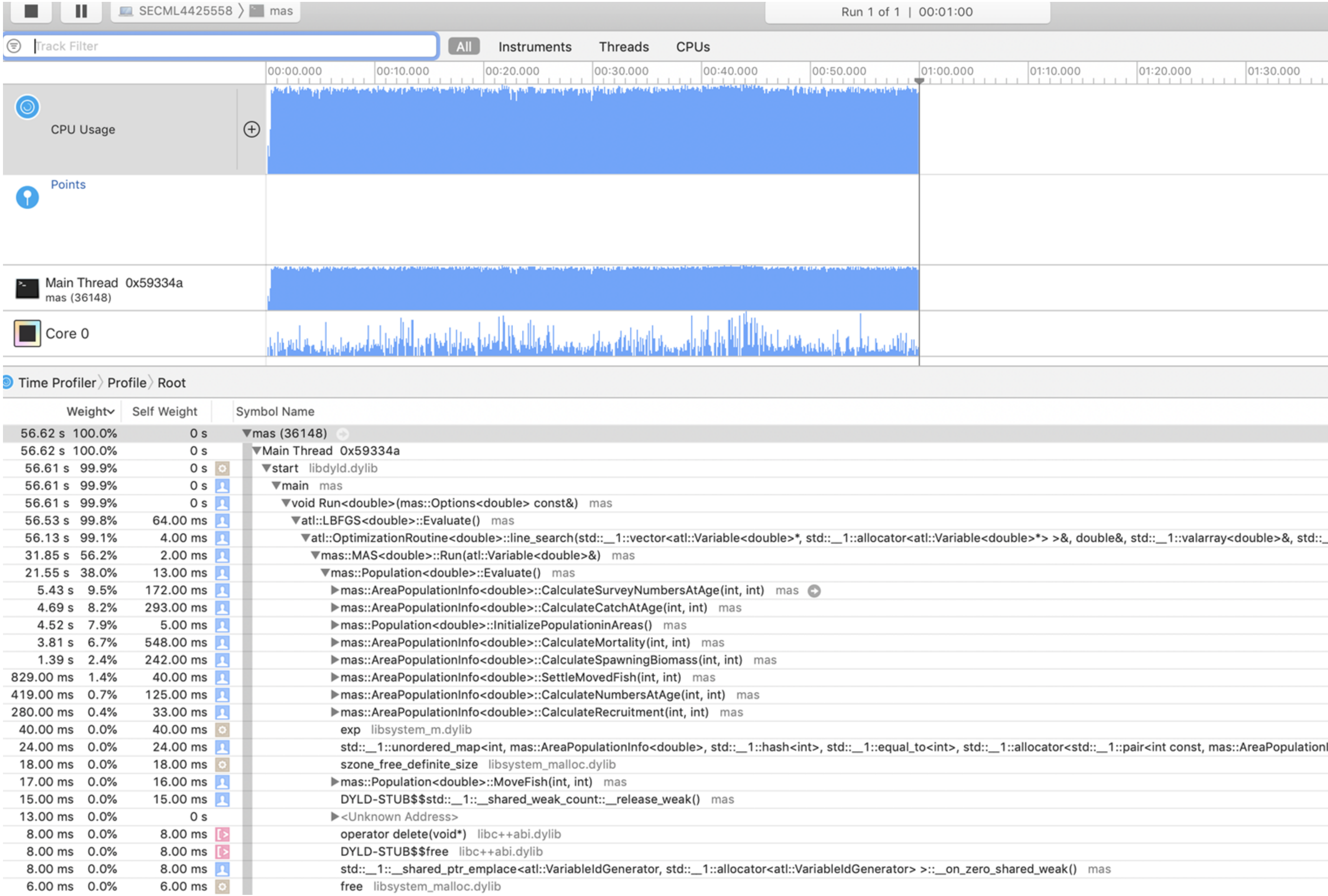
A profiler is an instrument used to perform dynamic analysis in a running application in order to obtain information on performance in regards to memory and CPU usage.

Tips For Better Code: *Use A* ***Profiler***

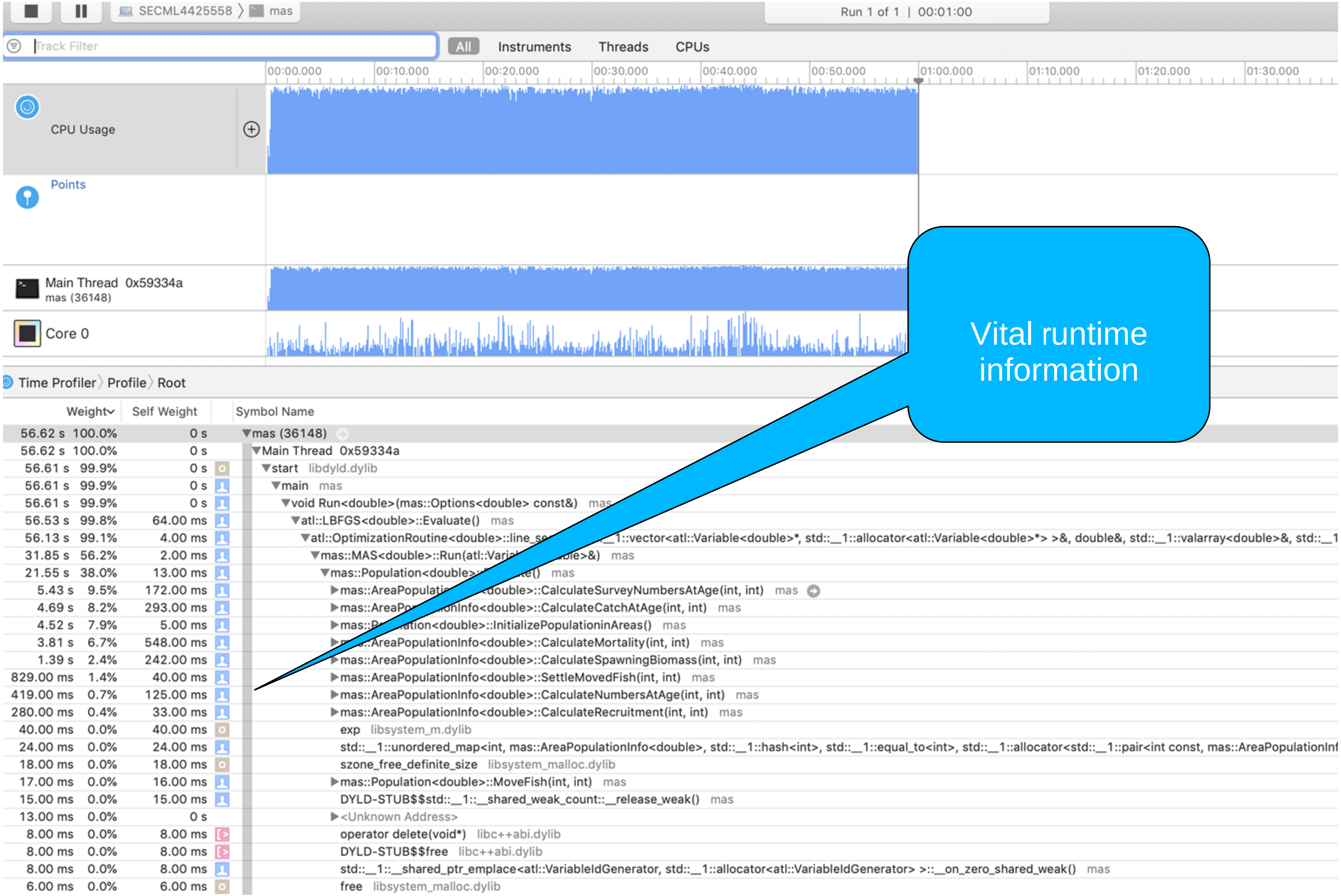
Why Use A Profiler?

- *Profilers allow you to find bottlenecks quickly.*
- *Find memory leaks.*
- *Collect statistics, such as memory usage, number of function calls, amount of time spent in a function, etc.*

Tips For Better Code: *Use A Profiler*



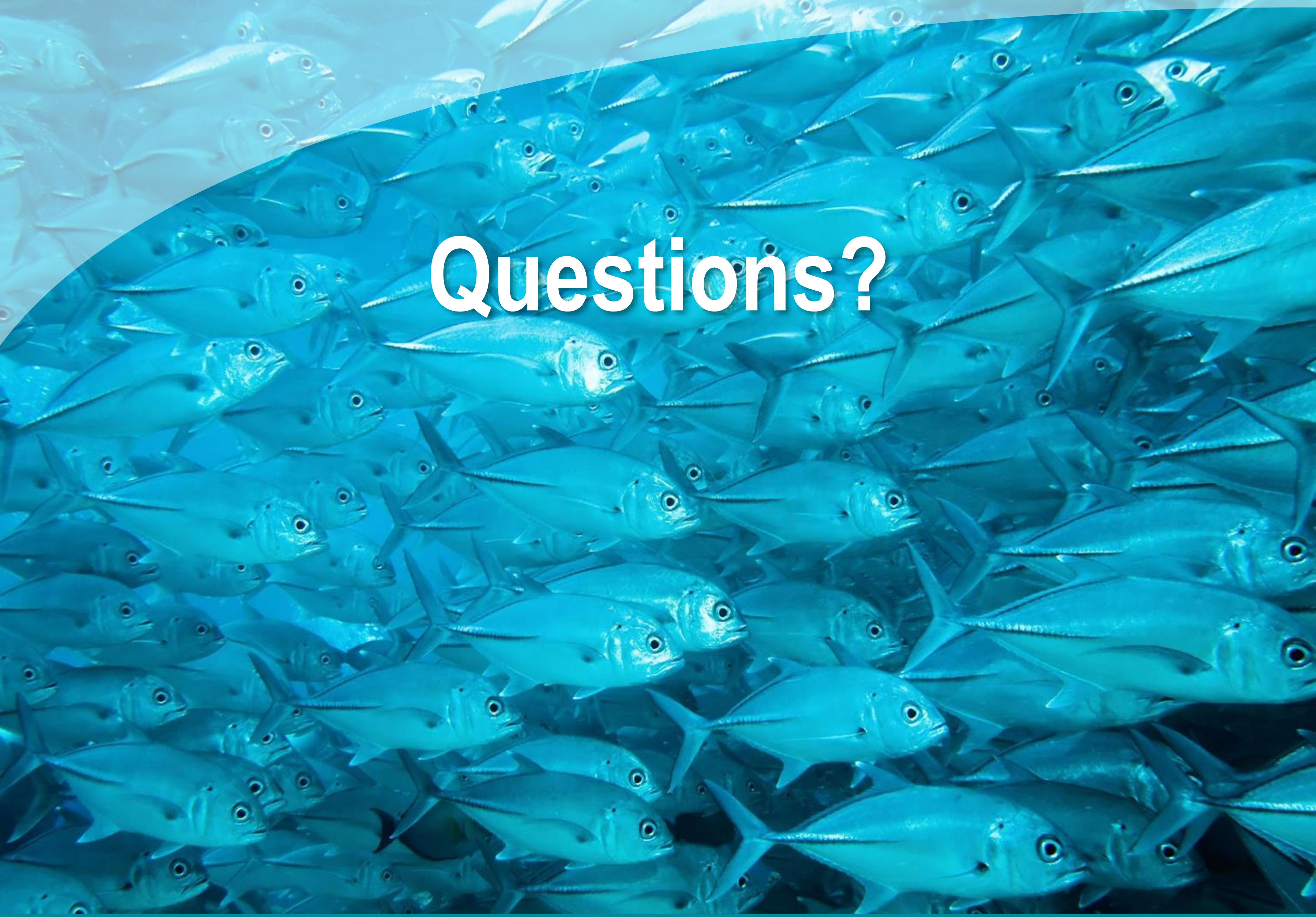
Tips For Better Code: *Use A Profiler*



NOAA FISHERIES

Summary

- *Use a disciplined/systematic approach to development.*
- *Use a modular design pattern.*
- *Keep the design simple.*
- *Use a structured or object-oriented programming paradigm.*
- *Remember the key concepts.*
    - *Modularity*
    - *Extensibility*
    - *Scalability*
    - *Incremental Development*
    - *Maintainability*
- *Use efficient, but readable code.*
- *Profile often.*


Questions?