Designing systems

Today we will dissect a program, e-mail.

- Goals:
  - Show how an e-mail system works
    - Interaction between devices.
    - Interaction between programs.
  - Show two design methods
  - Show the concept evolution.
- Why e-mail?
  - E-mail is a global application, running on 90% of all machines
  - You may take it for granted...

What is e-mail?

Human's view: E-mail is a system to allow people to exchange messages.

Computer's view: E-mail is a program.

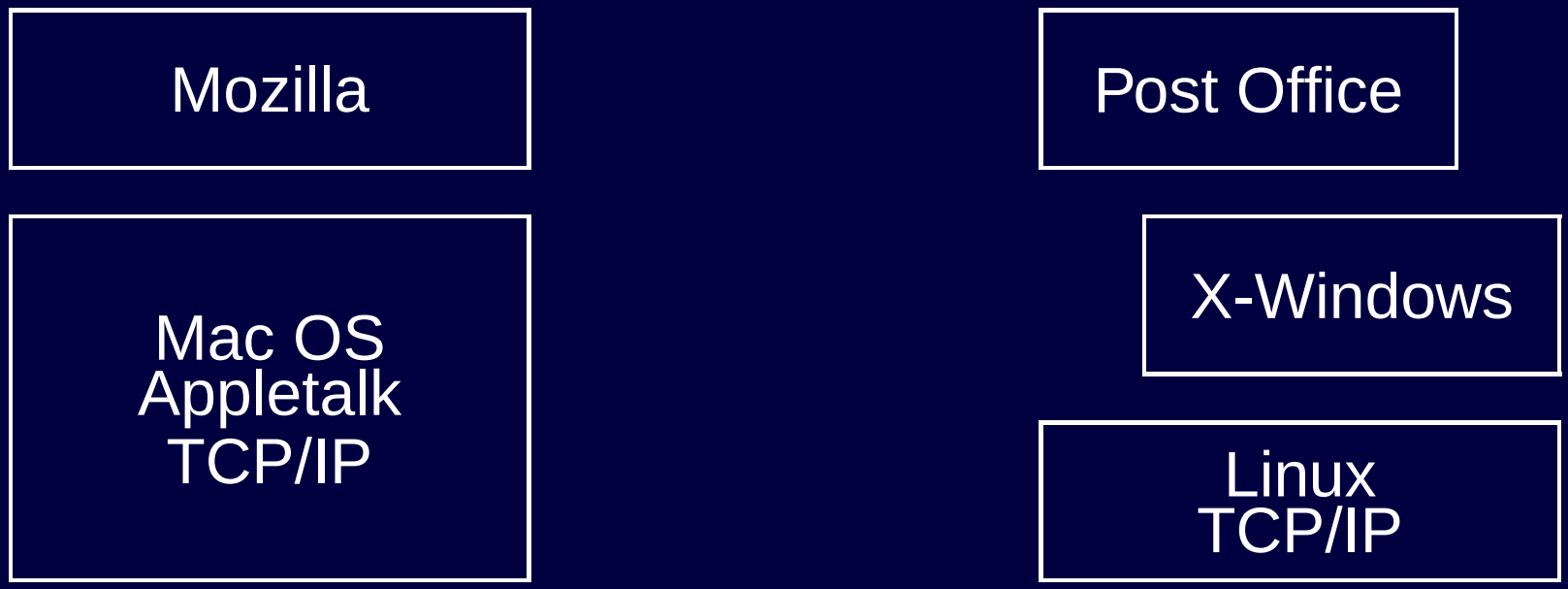
- It is an *application program*
- It is a program that serves the user.
- In contrast with a *System program*, which serves other programs.

Convenient to layer computers

Layers

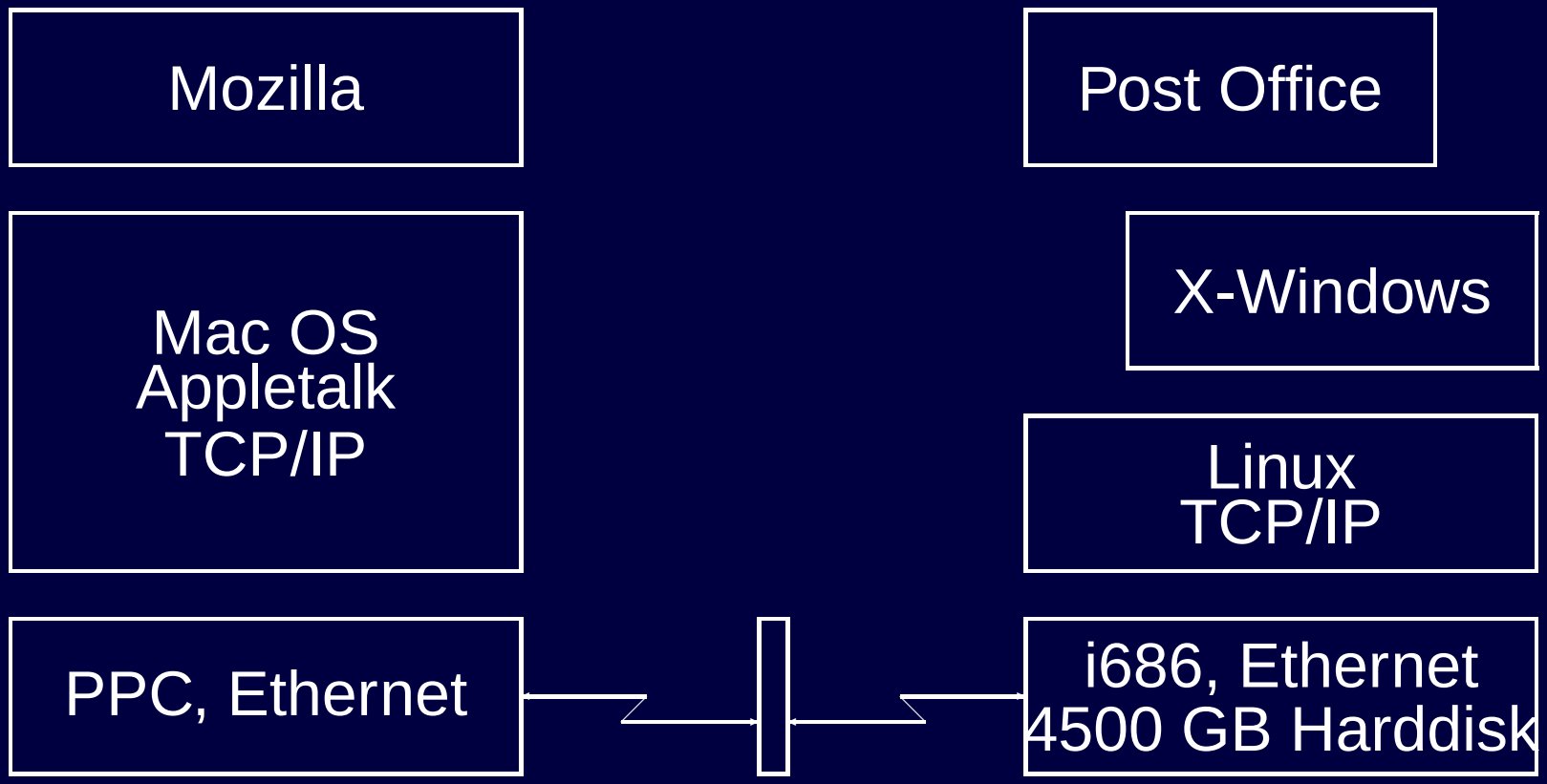
- Users view

Layers



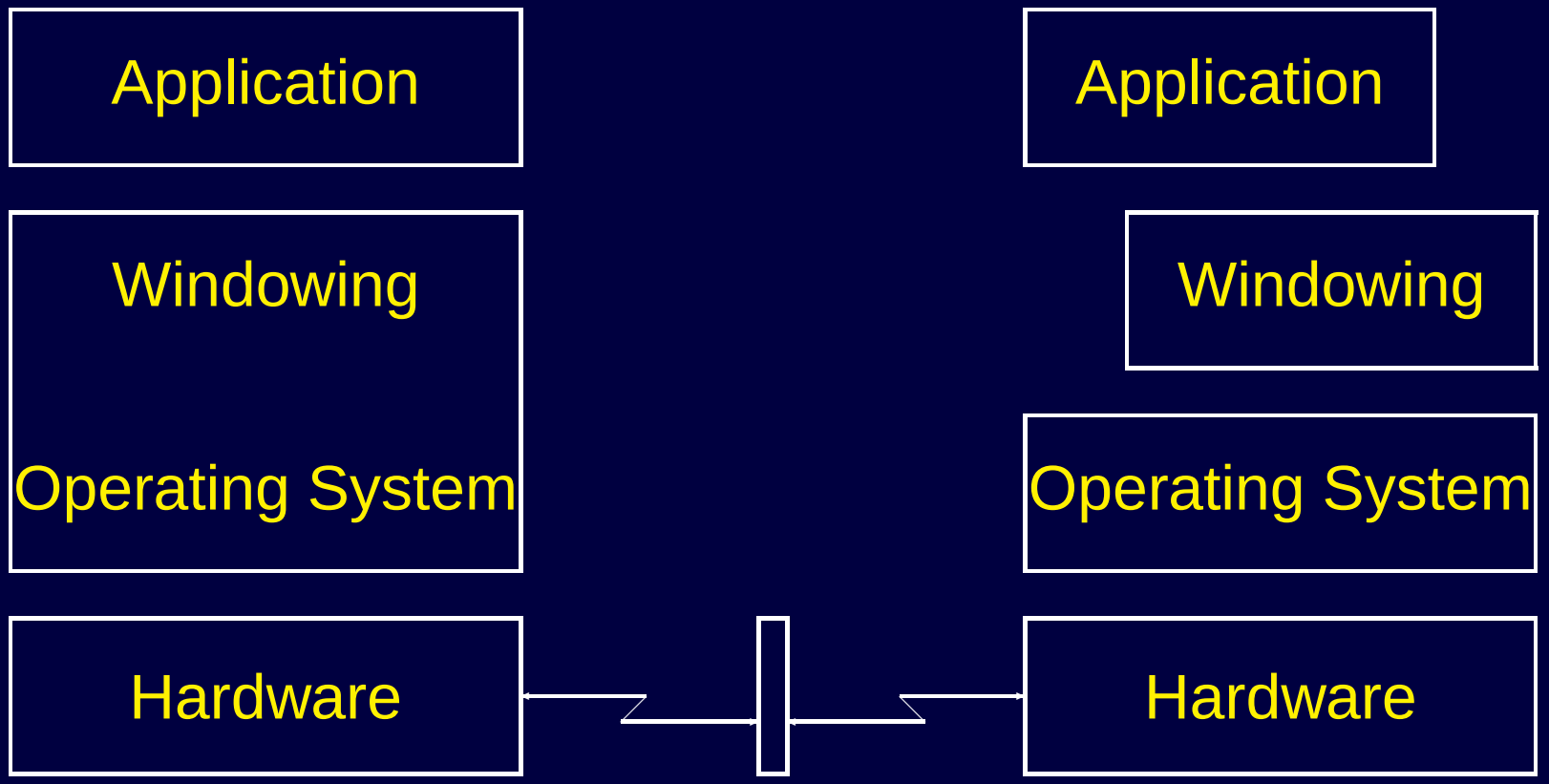
- There is something underneath

Layers



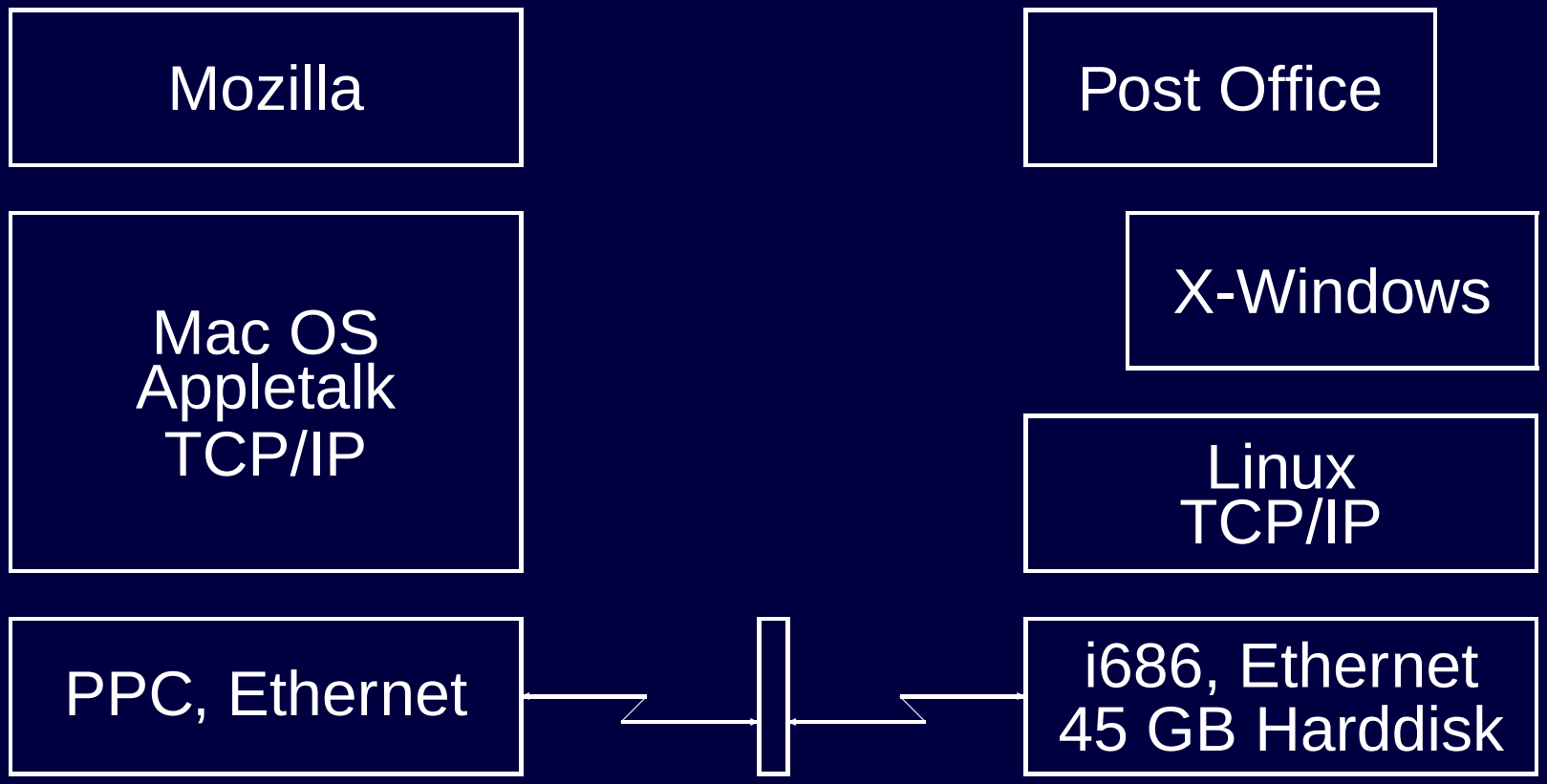
- And something underneath

Layers



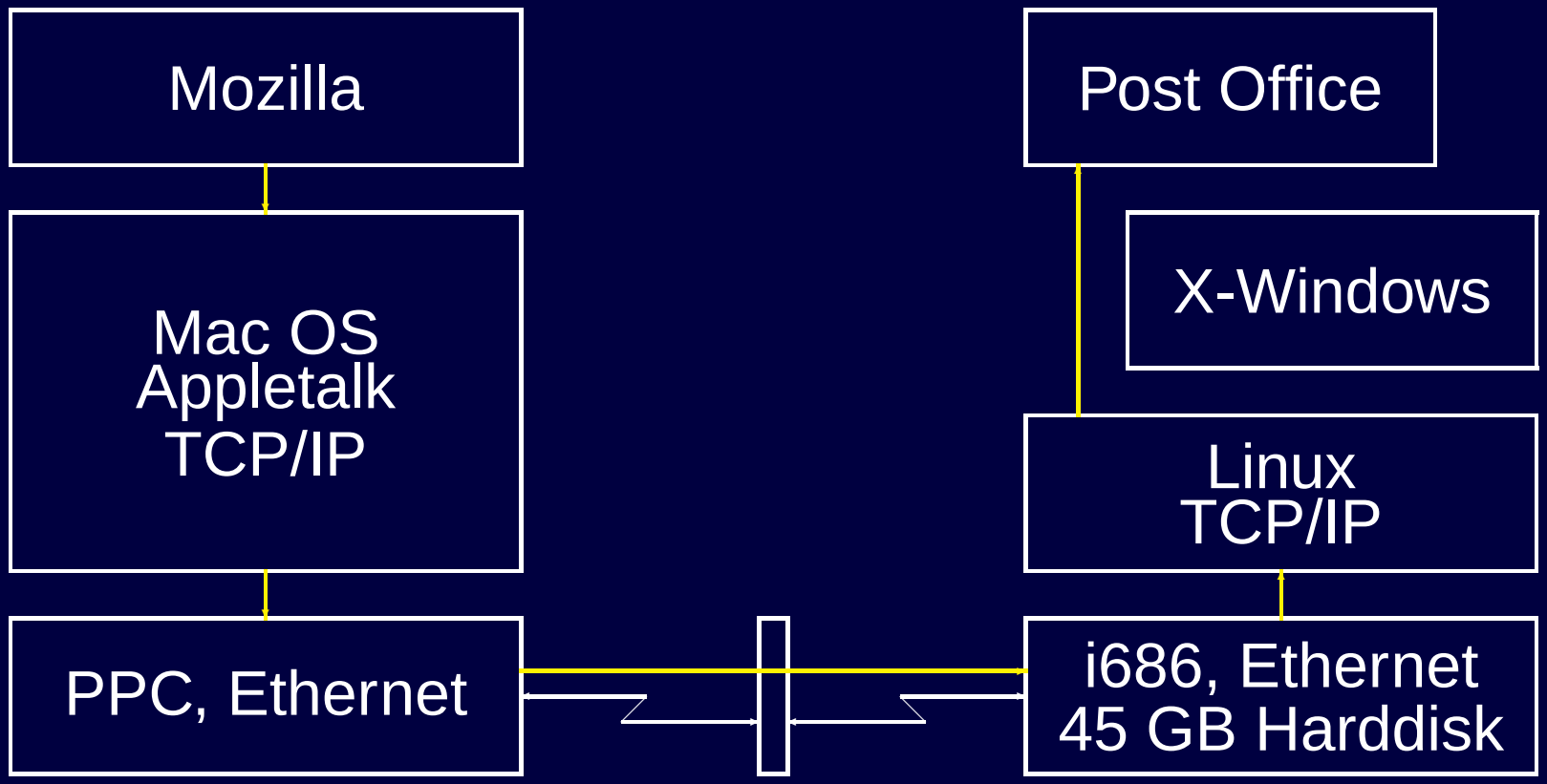
- General structure (layers!)

Layers



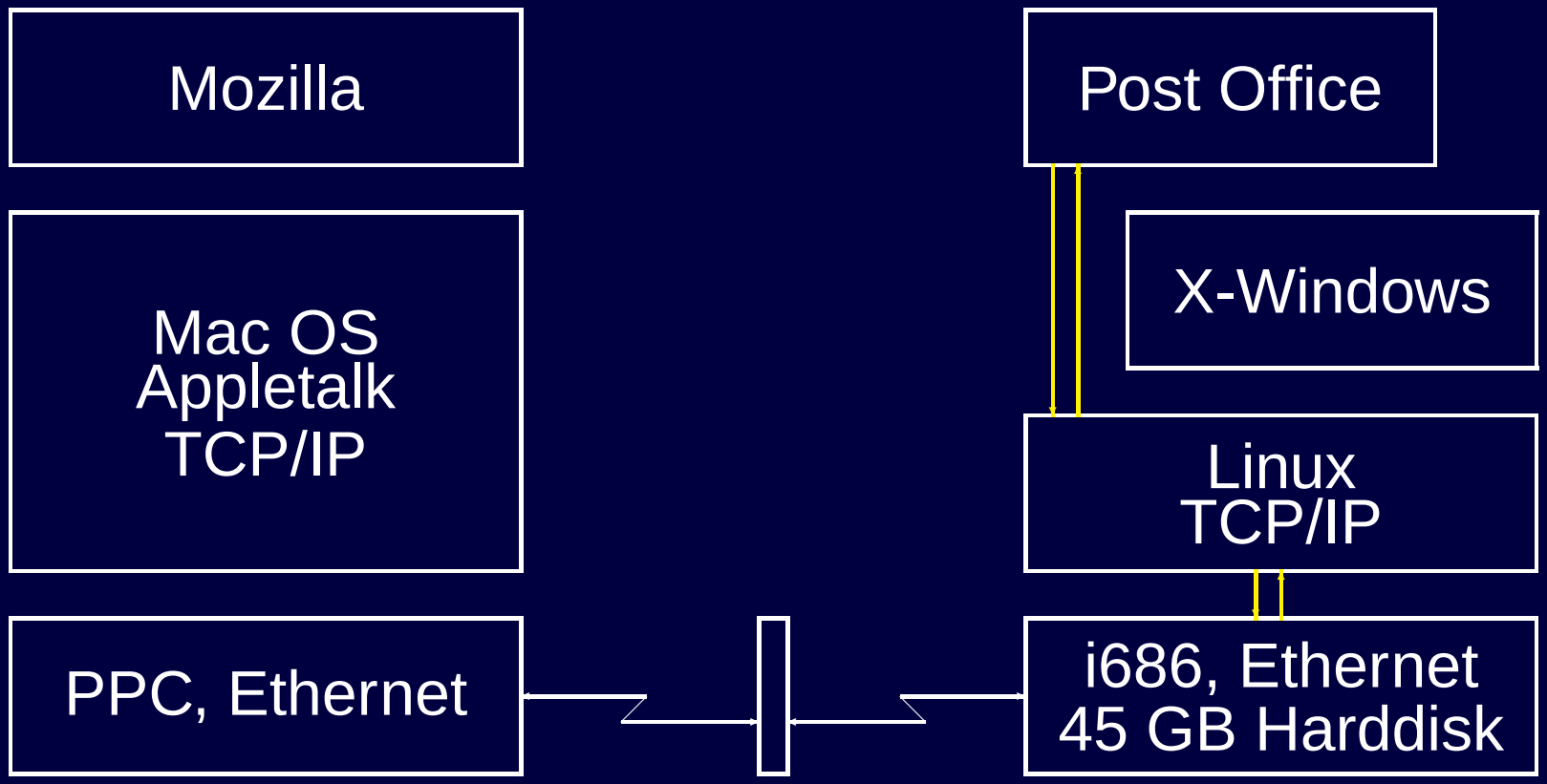
- How do the layers work?

Layers



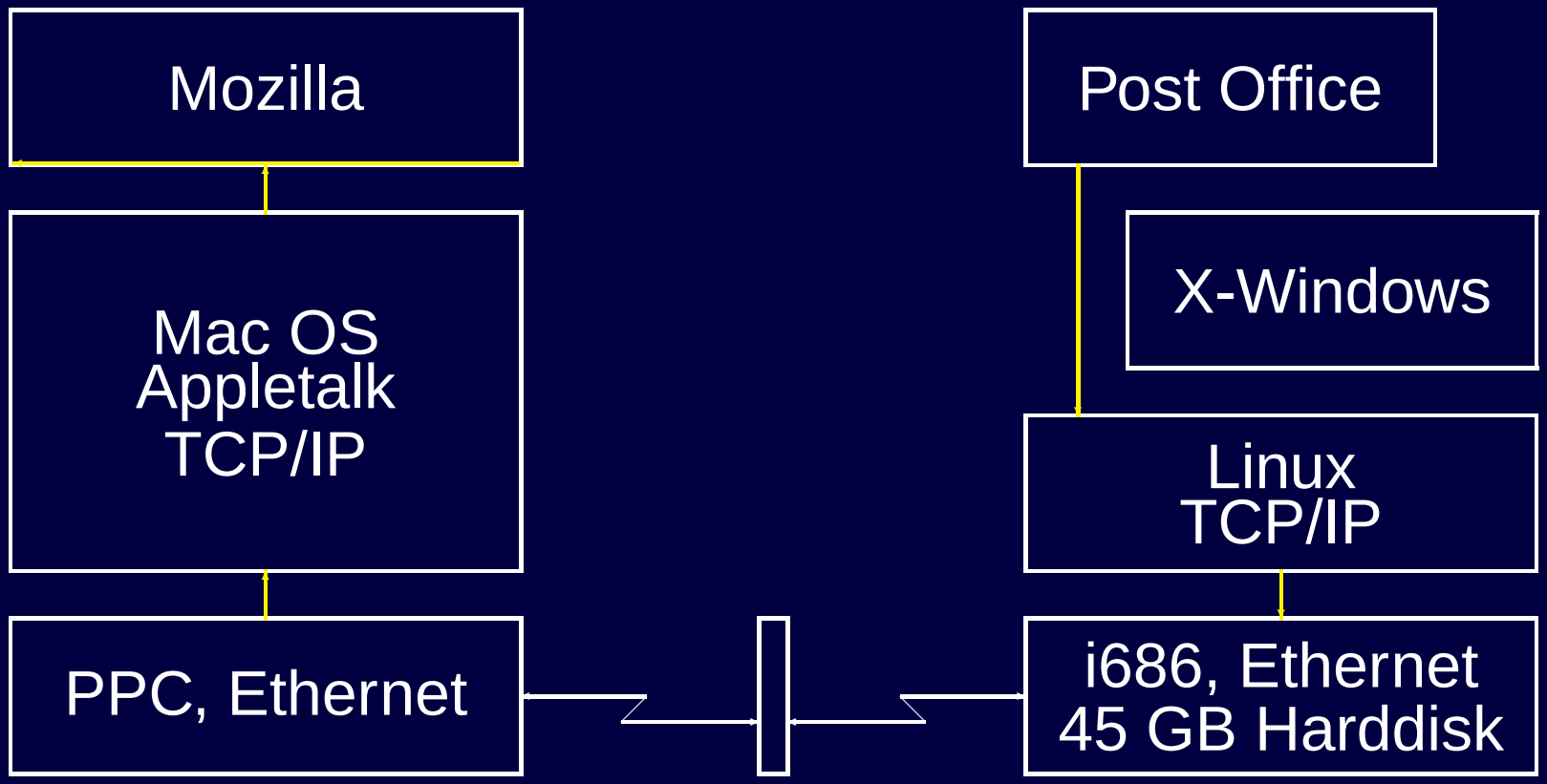
- Mail reader asks the post office

Layers



- Post office checks local mail repository

Layers



- Finally, Post office answers application

Rule 1 in problem solving

Computers are developed as independent parts

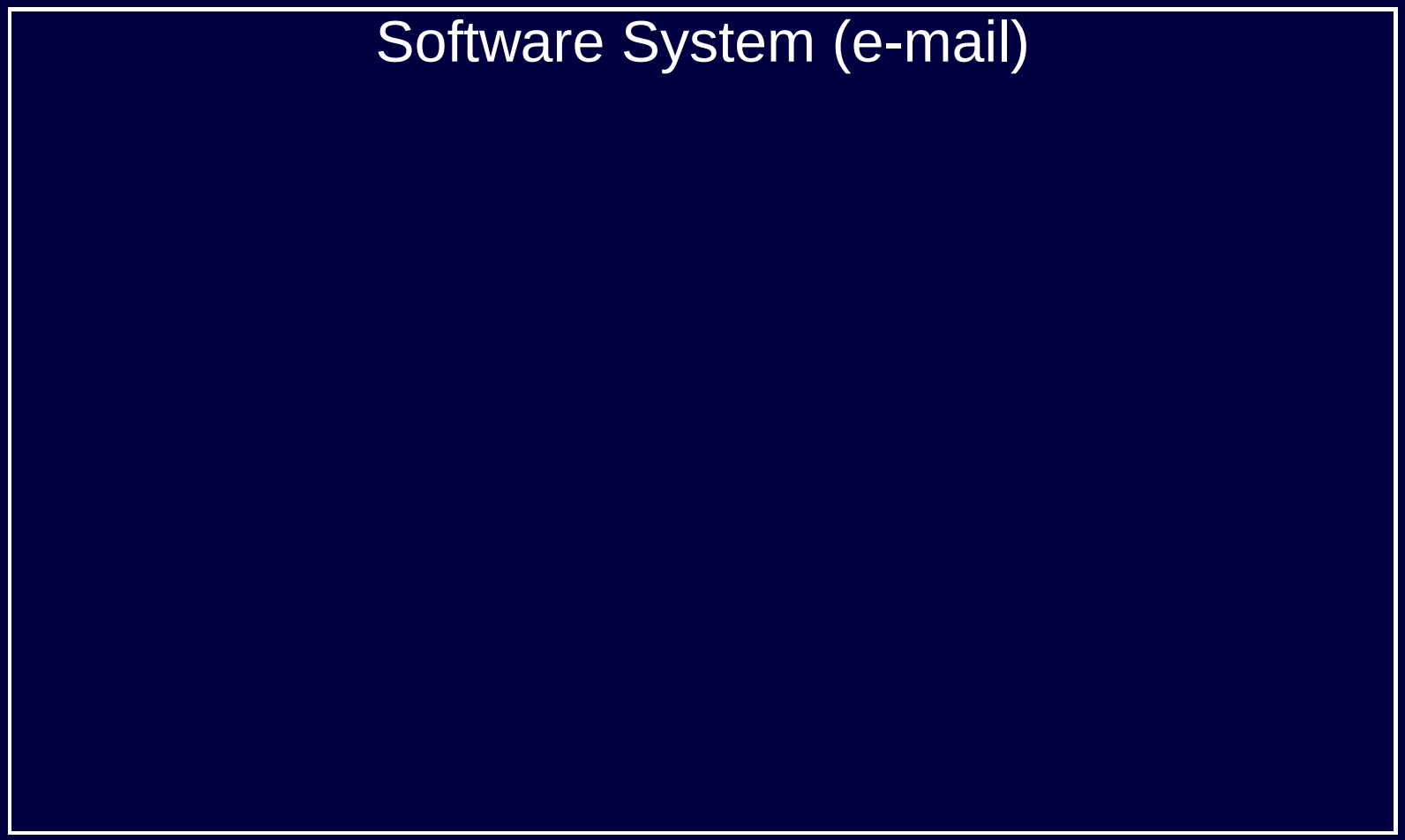
Divide and Conquer
or
Stepwise Refinement

If you try and solve a big problem (eg, develop e-mail)

1. Cut it up in parts (Reader, Post office, OS)
2. If the parts are trivial to solve: solve them.
3. If the parts are non trivial: apply divide and conquer again

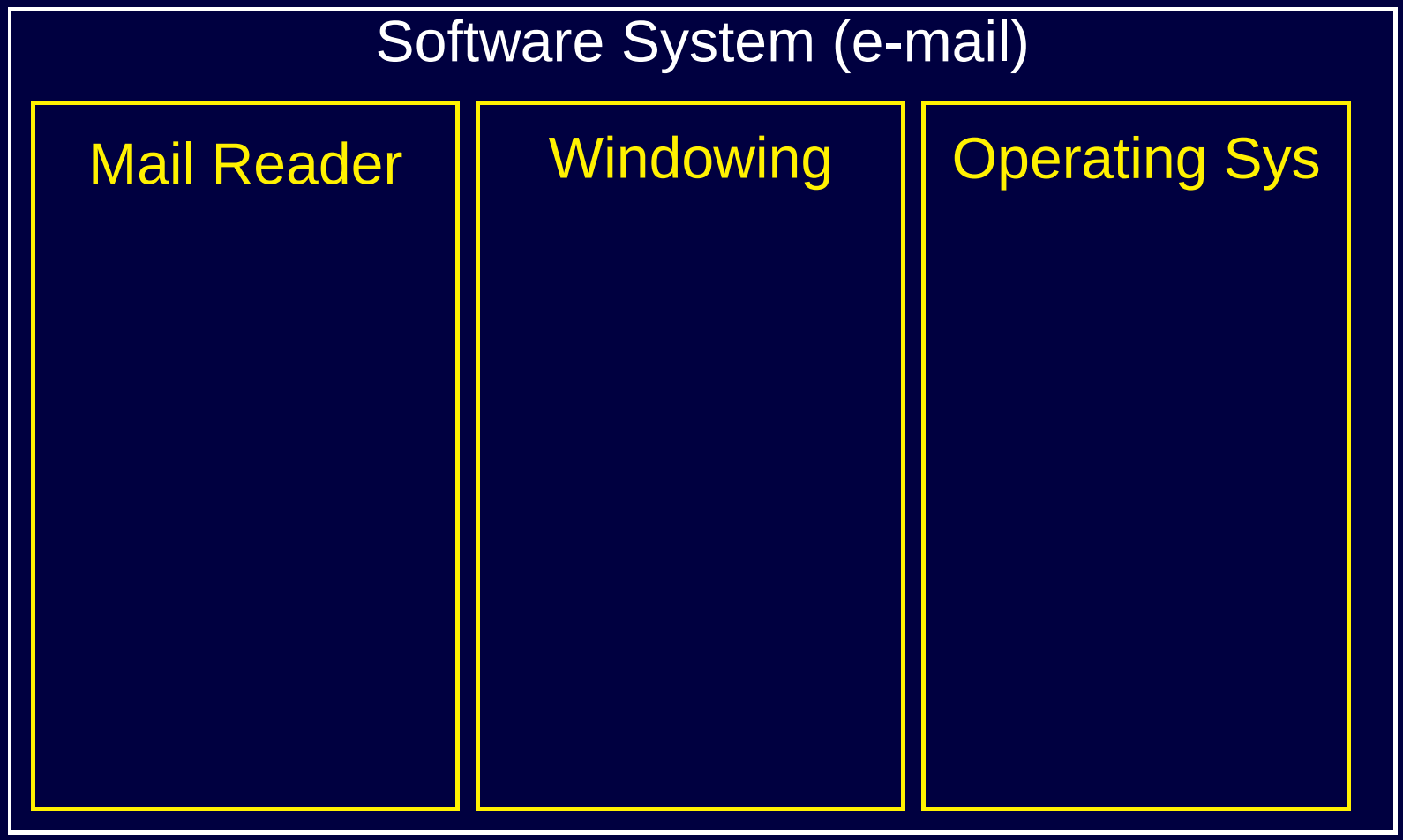
You will cut up your problem in smaller and smaller sub-sub-sub-problems until they can be solved trivially.

Hierarchical design: Software



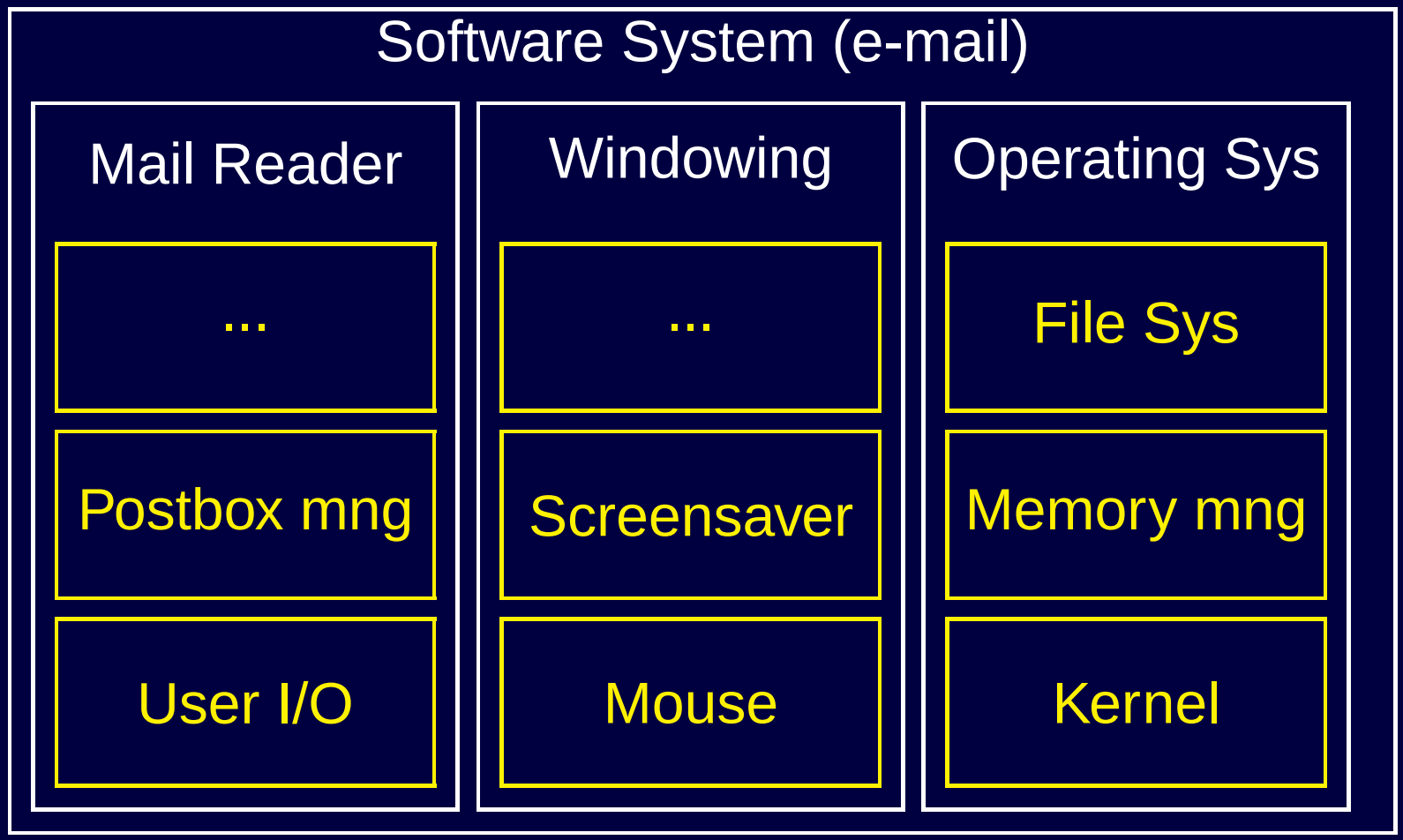
- From the outside: it seems a big system

Hierarchical design: Software



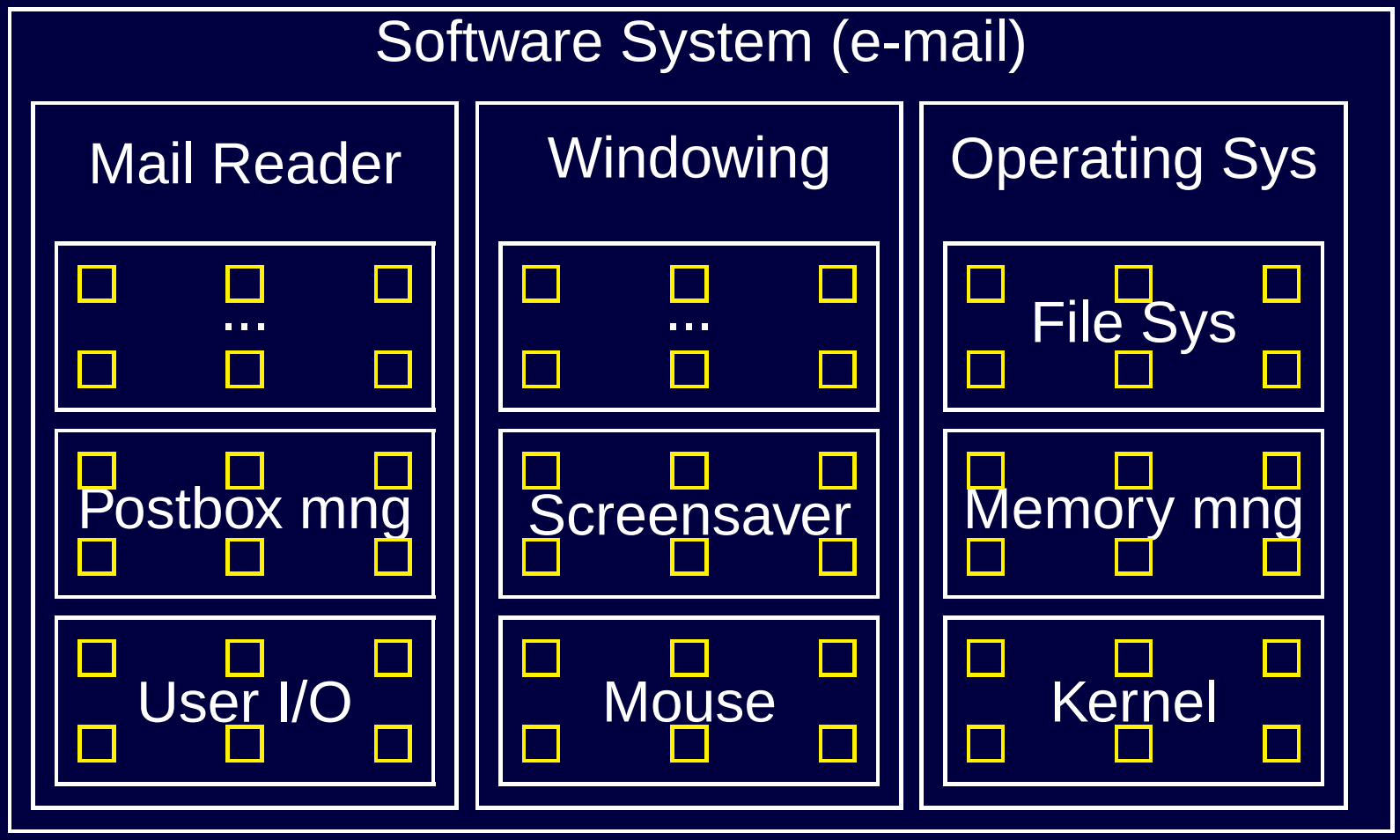
- The Software system is a collection of Programs

Hierarchical design: Software



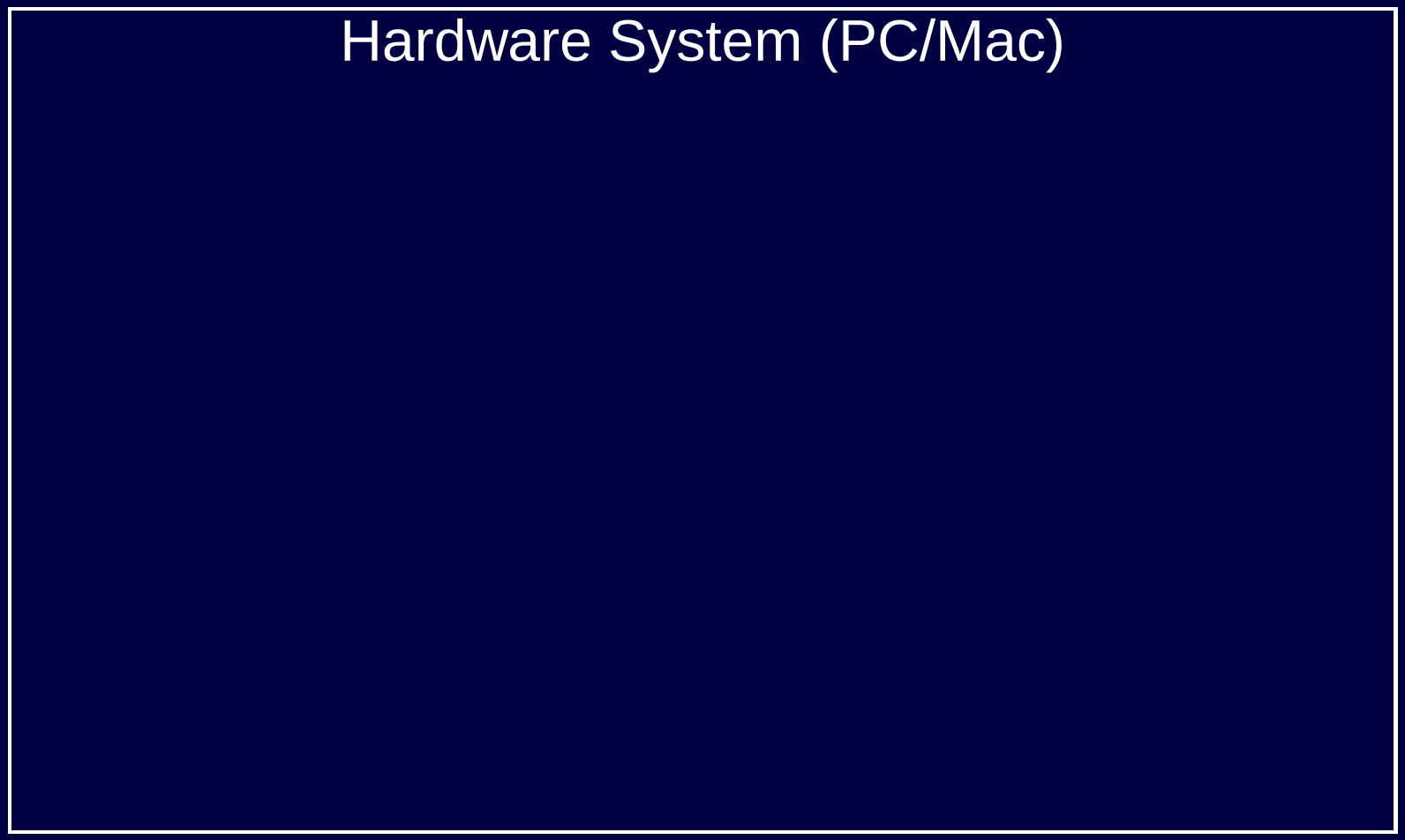
- Each program is a collection of modules

Hierarchical design: Software



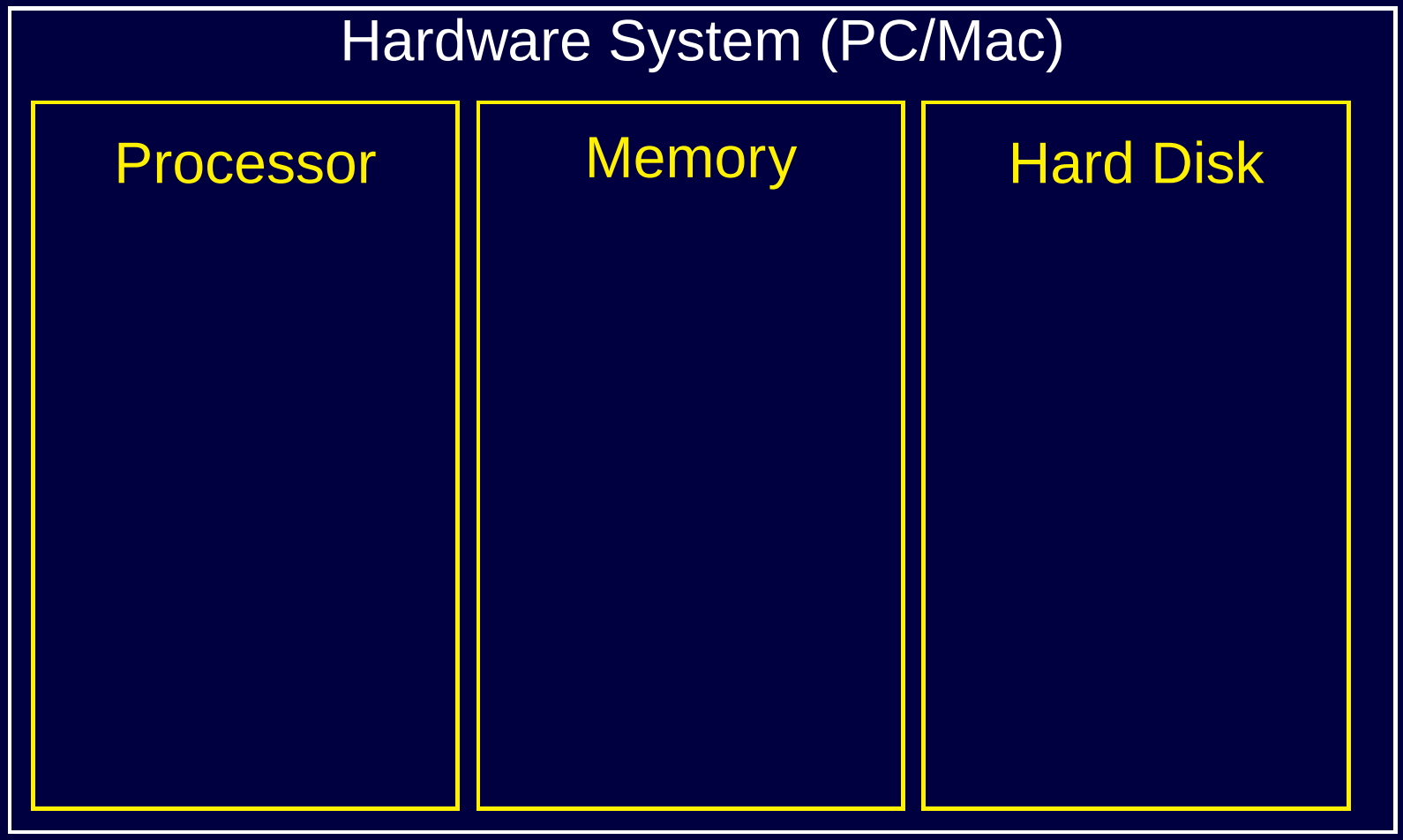
- Each module is a collection of functions

Hierarchical design: Hardware



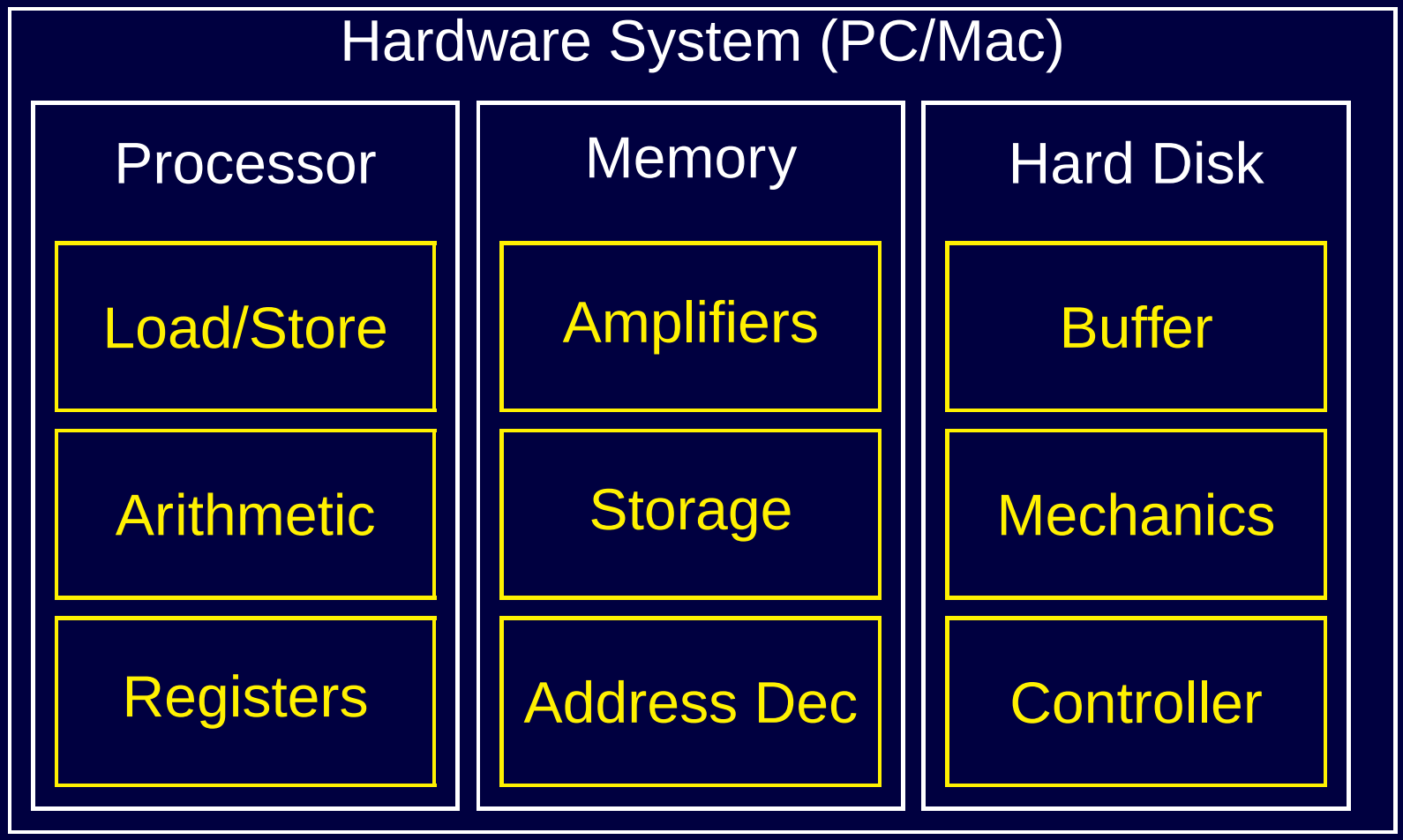
- From the outside: it seems a big system

Hierarchical design: Hardware



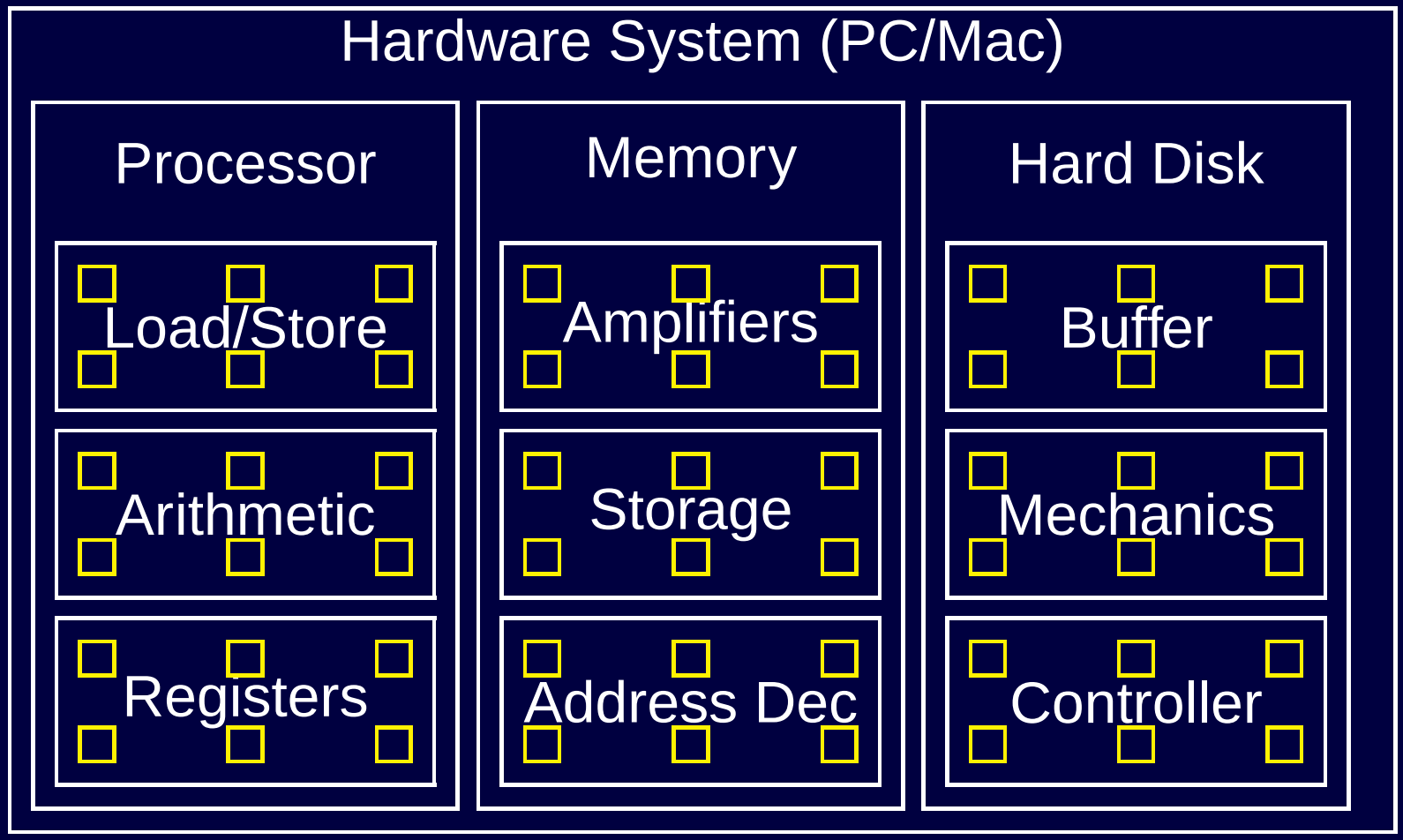
- The Hardware system is a collection of Components

Hierarchical design: Hardware



- We can go on dividing them into smaller parts

Hierarchical design: Hardware



- In the end, everything consists of transistors

Top-down versus Bottom-up

Divide and Conquer is a Top-Down design method.

- You start with what you want
- You break it up into small bits
- Until you can implement the bits

Other approach: Bottom-up design.

- You start with small bits
- You will put them together to form larger parts
- Until you end up with the thing that you wanted.

Problems

Problem with Top Down design:

- You must know how you break it up.
- You need intuition to tell you that this is the right way to divide the problem into sub problems.

Problem with Bottom Up design:

- Where do you start?
- usually used in an evolutionary way

  $\Rightarrow$ Systems start simple, functionality increases as time goes by.

Neither is perfect

$\Rightarrow$ You will do first some divide and conquer, and then some bottom-up...

Example of Evolutionary design

Evolutionary design: systems that grow organically.

- First E-mail systems (1970?) ran on one machine
  - No Postoffice.
  - No Windowing.
  - No Networking
- You could type:
- ```
  $ mail fraser
  This is a mail message
  .
  $
  ```
- This would deliver a mail message (no subject!) to fraser on the local system
- @ notation did not exist.
  ```

E-mail evolution - II

Second generation had primitive networking, and Subjects, around 1980.

You could type:

```
$ mail cwinl!uva!carol!fraser
Subject: This is a subject of the message
This is a mail message
.
$
```

This notation was known as the *bang* notation, using '!' to direct the mail

The sender had to know exactly how to send a message from A to B.

Note that peoples addresses depend on where the message originates!

@ notation did not exist.

E-mail evolution - III

Full blown networking:

- **`mail fraser`** would deliver to me.
  `@cs.bris.ac.uk`
- Wherever *you* are.

Windowing

- When window systems were developed, you just started your mail reader in a window. Still used (pine!).
- Then, late eighties, window based mail readers were developed.
  - Eudora, later Simeon, Netscape.
- Mail readers now accept graphics.

  $\Rightarrow$ Metamail.

E-mail evolution - III

Finally:

- Post Office (developed early nineties)

E-mail evolution summary:

- Each component designed top down.
- Evolution progressed bottom up.
  - $\Rightarrow$ One does not know what one wants (ever!)
  - $\Rightarrow$ Windows ? (1970?) Networks ? Post office ?

Evolution vs Top Down vs Bottom Up

Systems that grow by evolution:

- Work, but
- they are not very pretty (designwise)

In order to improve the structure, systems ought to be redesigned completely when they have evolved too far.

- First generation e-mail is good, second is ok, third generation works, but consists of original system plus 100 bolt-ons.
- Redesign system from scratch, taking the requirements of the third generation as your goal.
- Can evolve further after that.

The official design process

The design of a system, is part of the systems *Life-Cycle*

The Software Life-Cycle

1. Requirement analysis	Analyse the problem
2. Specification	Write down precisely what it should do
3. Design	using Divide and Conquer
4. Verification	using Divide and Conquer
5. Implementation	Using hardware and/or software
6. Testing	Try it in the lab
7. Delivery	Give it to the customer
8. Maintenance	Keep it working!

In this unit we will explore points 3, 5 and 6.

Summary

Top-down:

- Divide and Conquer.

Bottom-up:

- Start by building the smallest component

Evolution:

- Systems aren’t static, requirements change, systems are changed, systems need to be redesigned completely when the structure is lost.

Programming

Programming:

- Designing Software.
- Coding it in a form suitable for a computer to understand it.

Design:

- Ask for the material
- Ask for the dimensions (span, #columns, width, ...)
- Calculate the strength using equations X
- Calculate possible resonance using equations Y

Code:

- Load of squiglly brackets, semicolons, cryptic statements.
- *Commands*, or *Rules* that tell the computer what to do.

Example program

```
int main( void ) {
  printf( "I write garbage\n" ) ;
  return 0 ;
}
```

What does this mean?

- **`int main( void )`**: between **`{ }`** comes the main part of the program
- **`printf( ... ) ;`**: print everything between **`()`** on the screen
- **`return 0;`**: the main program is ready, return control to the user.

Actually:

- **`main`** is a *function*
  - A program may consist of one or more functions
- **`printf( ... ) ;`** is a *statement*.
  - A function may consist of zero or more statements.

Example program - II

```
int main( void ) {
  printf( "I write garbage\n" ) ;
  printf( "I print %f cabbage\n", 5.0 ) ;
  printf( "I like brocolli %f\n", 2.0/3.0 ) ;
  printf( "Ok, it *is* late in the evening\n" ) ;
  return 0 ;
}
```

The four lines **`printf( ... ) ;`** are four *statements*.

- Each of these will be *executed* in turn
- The program will print “I write ... evening”.

Can we make a program with multiple functions?

Execute it...

```
int main( void ) {
  printf( "I write garbage\n" ) ;
  printf( "I print %f cabbage\n", 5.0 ) ;
  printf( "I like brocolli %f\n", 2.0/3.0 ) ;
  printf( "Ok, it *is* late in the evening\n" ) ;
  return 0 ;
}
```

Execute it...

⇒
```
int main( void ) {
  printf( "I write garbage\n" ) ;
  printf( "I print %f cabbage\n", 5.0 ) ;
  printf( "I like brocolli %f\n", 2.0/3.0 ) ;
  printf( "Ok, it *is* late in the evening\n" ) ;
  return 0 ;
}
```

Execute it...

```
  int main( void ) {
⇒   printf( "I write garbage\n" ) ;
    printf( "I print %f cabbage\n", 5.0 ) ;
    printf( "I like brocolli %f\n", 2.0/3.0 ) ;
    printf( "Ok, it *is* late in the evening\n" ) ;
    return 0 ;
  }
```

Execute it...

```
int main( void ) {
  printf( "I write garbage\n" ) ;
⇒   printf( "I print %f cabbage\n", 5.0 ) ;
  printf( "I like brocolli %f\n", 2.0/3.0 ) ;
  printf( "Ok, it *is* late in the evening\n" ) ;
  return 0 ;
}
```

```
I write garbage
```

Execute it...

```
int main( void ) {
  printf( "I write garbage\n" ) ;
  printf( "I print %f cabbage\n", 5.0 ) ;
⇒ printf( "I like brocolli %f\n", 2.0/3.0 ) ;
  printf( "Ok, it *is* late in the evening\n" ) ;
  return 0 ;
}
```

```
I write garbage
I print 5.000000 cabbage
```

Execute it...

```
int main( void ) {
  printf( "I write garbage\n" ) ;
  printf( "I print %f cabbage\n", 5.0 ) ;
  printf( "I like brocolli %f\n", 2.0/3.0 ) ;
⇒   printf( "Ok, it *is* late in the evening\n" ) ;
  return 0 ;
}
```

```
I write garbage
I print 5.000000 cabbage
I like brocolli 0.666667
```

Execute it...

```
int main( void ) {
  printf( "I write garbage\n" ) ;
  printf( "I print %f cabbage\n", 5.0 ) ;
  printf( "I like brocolli %f\n", 2.0/3.0 ) ;
  printf( "Ok, it *is* late in the evening\n" ) ;
⇒   return 0 ;
}
```

```
I write garbage
I print 5.000000 cabbage
I like brocolli 0.666667
Ok, it *is* late in the evening
```

More complicated Program

```
double cylinder( double r, double h ) {
  return 3.1415926535 * r * r * h ;
}
int main( void ) {
  printf("Cylinder(1,1)=%f\n",  cylinder(1,1) ) ;
  printf("Cylinder(2,1)=%f\n",  cylinder(2,1) ) ;
  printf("Cylinder(10,25)=%f\n",cylinder(10,25));
  return 0 ;
}
```

- This program has two functions: **`main`** and **`cylinder`**
- The line **`double cylinder( double r, double h )`** says: between **`{}`** I will define a function “cylinder”. I made up that name, it calculates the contents of a cylinder with radius **`r`**, and height **`h`**
- **`double r`** means: **`r`** is a real number

Execute it...

```
double cylinder( double r, double h ) {
  return 3.1415926535 * r * r * h ;
}

int main( void ) {
  printf("Cylinder(1,1)=%f\n",  cylinder(1,1) ) ;
  printf("Cylinder(2,1)=%f\n",  cylinder(2,1) ) ;
  printf("Cylinder(10,25)=%f\n",cylinder(10,25));
  return 0 ;
}
```

Execute it...

```
double cylinder( double r, double h ) {
  return 3.1415926535 * r * r * h ;
}

int main( void ) {
⇒   printf("Cylinder(1,1)=%f\n",  cylinder(1,1) ) ;
    printf("Cylinder(2,1)=%f\n",  cylinder(2,1) ) ;
    printf("Cylinder(10,25)=%f\n",cylinder(10,25));
    return 0 ;
}
```

Execute it...

```
⇒ double cylinder( double r, double h ) {
    return 3.1415926535 * r * r * h ;
  }                       1   1   1

  int main( void ) {
    printf("Cylinder(1,1)=%f\n",  cylinder(1,1) ) ;
    printf("Cylinder(2,1)=%f\n",  cylinder(2,1) ) ;
    printf("Cylinder(10,25)=%f\n",cylinder(10,25));
    return 0 ;
  }
```

Execute it...

```
  double cylinder( double r, double h ) {
⇒   return 3.1415926535 * r * r * h ;
  }                         1   1   1

  int main( void ) {
    printf("Cylinder(1,1)=%f\n",  cylinder(1,1) ) ;
    printf("Cylinder(2,1)=%f\n",  cylinder(2,1) ) ;
    printf("Cylinder(10,25)=%f\n",cylinder(10,25));
    return 0 ;
  }
```

Execute it...

```
double cylinder( double r, double h ) {
  return 3.1415926535 * r * r * h ;
}

int main( void ) {
  printf("Cylinder(1,1)=%f\n",  cylinder(1,1) ) ;
  printf("Cylinder(2,1)=%f\n",  cylinder(2,1) ) ;
  printf("Cylinder(10,25)=%f\n",cylinder(10,25));
  return 0 ;
}
```

Execute it...

```
double cylinder( double r, double h ) {
  return 3.1415926535 * r * r * h ;
}

int main( void ) {
  printf("Cylinder(1,1)=%f\n",  cylinder(1,1) ) ;
⇒ printf("Cylinder(2,1)=%f\n",  cylinder(2,1) ) ;
  printf("Cylinder(10,25)=%f\n",cylinder(10,25));
  return 0 ;
}
```

```
Cylinder(1,1)=3.141593
```

Execute it...

```
⇒ double cylinder( double r, double h ) {
    return 3.1415926535 * r * r * h ;
  }                       2   2   1

  int main( void ) {
    printf("Cylinder(1,1)=%f\n",  cylinder(1,1) ) ;
    printf("Cylinder(2,1)=%f\n",  cylinder(2,1) ) ;
    printf("Cylinder(10,25)=%f\n",cylinder(10,25));
    return 0 ;
  }
```

```
Cylinder(1,1)=3.141593
```

Execute it...

```
  double cylinder( double r, double h ) {
⇒   return 3.1415926535 * r * r * h ;
  }                         2   2   1

  int main( void ) {
    printf("Cylinder(1,1)=%f\n",  cylinder(1,1) ) ;
    printf("Cylinder(2,1)=%f\n",  cylinder(2,1) ) ;
    printf("Cylinder(10,25)=%f\n",cylinder(10,25));
    return 0 ;
  }
```

```
Cylinder(1,1)=3.141593
```

Execute it...

```
double cylinder( double r, double h ) {
  return 3.1415926535 * r * r * h ;
}

int main( void ) {
  printf("Cylinder(1,1)=%f\n", ⇓cylinder(1,1) ) ;
  printf("Cylinder(2,1)=%f\n",  cylinder(2,1) ) ;
  printf("Cylinder(10,25)=%f\n",cylinder(10,25));
  return 0 ;
}
```

```
Cylinder(1,1)=3.141593
```

Execute it...

```
double cylinder( double r, double h ) {
  return 3.1415926535 * r * r * h ;
}

int main( void ) {
  printf("Cylinder(1,1)=%f\n",  cylinder(1,1) ) ;
  printf("Cylinder(2,1)=%f\n",  cylinder(2,1) ) ;
⇒ printf("Cylinder(10,25)=%f\n",cylinder(10,25));
  return 0 ;
}
```

```
Cylinder(1,1)=3.141593
Cylinder(2,1)=12.566371
```

Execute it...

```
⇒ double cylinder( double r, double h ) {
    return 3.1415926535 * r * r * h ;
  }                          10  10  25

  int main( void ) {
    printf("Cylinder(1,1)=%f\n",  cylinder(1,1) ) ;
    printf("Cylinder(2,1)=%f\n",  cylinder(2,1) ) ;
    printf("Cylinder(10,25)=%f\n",cylinder(10,25));
    return 0 ;
  }
```

```
Cylinder(1,1)=3.141593
Cylinder(2,1)=12.566371
```

Execute it...

```
  double cylinder( double r, double h ) {
⇒   return 3.1415926535 * r * r * h ;
  }                          10  10  25

  int main( void ) {
    printf("Cylinder(1,1)=%f\n",  cylinder(1,1) ) ;
    printf("Cylinder(2,1)=%f\n",  cylinder(2,1) ) ;
    printf("Cylinder(10,25)=%f\n",cylinder(10,25));
    return 0 ;
  }
```

```
Cylinder(1,1)=3.141593
Cylinder(2,1)=12.566371
```

Execute it...

```
double cylinder( double r, double h ) {
  return 3.1415926535 * r * r * h ;
}

int main( void ) {
  printf("Cylinder(1,1)=%f\n",  cylinder(1,1) ) ;
  printf("Cylinder(2,1)=%f\n",  cylinder(2,1) ) ;
  printf("Cylinder(10,25)=%f\n",cylinder(10,25));
  return 0 ;                          ⇑
}
```

```
Cylinder(1,1)=3.141593
Cylinder(2,1)=12.566371
```

Execute it...

```
double cylinder( double r, double h ) {
  return 3.1415926535 * r * r * h ;
}

int main( void ) {
  printf("Cylinder(1,1)=%f\n",  cylinder(1,1) ) ;
  printf("Cylinder(2,1)=%f\n",  cylinder(2,1) ) ;
  printf("Cylinder(10,25)=%f\n",cylinder(10,25));
⇒   return 0 ;
}
```

```
Cylinder(1,1)=3.141593
Cylinder(2,1)=12.566371
Cylinder(10,25)=7853.98163
```

Execution model

The programs I showed you are written in C.

- Execution model of C:
  - At any time, a statement is executed,
  - We advance the pointer to the next statement.
- These statements do something
  - Calculate the contents of the cylinder
  - Print something on the screen.

This is known as procedural programming.

Paradigms

Different forms of programming are known as different paradigms

1. Procedural programming
   - Gives commands to do things
   - These commands change the state

2. Declarative programming
   - Gives definitions of operations
   - Tell the system what you want to know, and it will figure it out

3. Object Oriented programming
   - Like procedural, but has operations on objects only.
   - Operations change the state of an object.

Example Languages

Procedural

- C, Pascal, Basic, Fortran, Cobol, Algol, ...

Declarative

- Haskell, Prolog, Miranda, Lisp, Gödel, ...

Object Oriented

- Java, Simula, SmallTalk, Clu, C++, Objective C, ...

C and Haskell are taught in this course (elementary only, you will have to teach yourself the more advanced aspects)

Java is taught in COMS10001.

Why Computer Languages

Computers are stupid.

- They do not “understand” things
- They have to be told very precisely what to do.
- How many of you did not type the space between **`chmod a+x`** and `..`

Natural languages are ambiguous

- If you programmed a computer in a natural language it would never do what you want it to do.

Programming languages are designed to be unambiguous.

- Every program which follows the syntax and grammar has an unambigously defined meaning.

Ambiguities

Natural languages are ambiguous

- I publish Bristol's largest quality free newspaper

Ambiguities

Natural languages are ambiguous

- I publish Bristol's largest quality (free newspaper)

Ambiguities

Natural languages are ambiguous

- I publish Bristol's largest (quality free) newspaper

Ambiguities

Natural languages are ambiguous

- I publish Bristol's largest quality free newspaper

Programming languages are not ambiguous.

- They are defined precisely
- The meaning of each program is defined precisely.

$\Rightarrow$ Priority of operators is defined. C: `2+3*4` MEANS `2+(3*4)`.

$\Rightarrow$ Associativity is defined. C: `2-3-4` MEANS `(2-3)-4`.

Syntax, Grammar, Semantics of English

Syntax of English:

- use the Roman letters.
- separate words by spaces.

Grammar of English

- A correct sentence is of the form Noun Verb Adjective, or of the form Noun Verb Noun, or ...

The Semantics of English defines the meaning:

- “I like broccoli” has a meaning (it means that I feel happy when I eat this green stuff)
- “I wonder broccoli” has no meaning (even though it is grammatically correct)

Syntax, Grammar, Semantics of C

The Syntax of C:

- use the ASCII character set.
- words are separated by space, newline, tab, or any non digit/character.

The Grammar of C says that

- a program shall be composed of functions.
- a function shall be composed of statements.

The semantics of C defines the meaning:

- "**`a=3 ; b=6 ; z=b*a ;`**" has a meaning **`z`** becomes 18.
- "**`a=0 ; b=6 ; z=b/a ;`**" has no meaning, even though it is grammatically correct.

Correct / Incorrect programs

If you type a program which does not confirm to the syntax/grammar:

- Compiler will refuse it, will tell you where your program is wrong.

$\Rightarrow$ You will have to take typos out.

- Easy, just do what the compiler says (semicolon expected, missing `)`, ...)

A program which is syntactically correct, it is not necessarily right:

- It can be semantically wrong.

$\Rightarrow$ You will have to take "bugs" out; known as debugging.

- This is an art: the computer won't tell you why it doesn't work.
- Suppose, you type `i=i;` instead of `i=1;`, both are semantically, grammatically, and syntactically correct...

Summary

- A program is a collection of *functions*
- A function is a collection of *statements*
  - Each statement ends with a `;` (semi-colon)
  - Statements of a function are enclosed in **`{}`** (curly braces).
- Execution of the program starts at the *main* function
- A function is executed by sequentially executing its statements
  - Functions can have parameters, *unknowns* in mathematics.
  - Functions produce a value by means of **`return`** *expression*.
- **`printf("Bla %d\n",`** *expression***`);`** prints an expression on your screen.