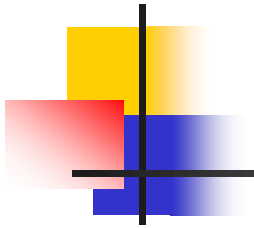
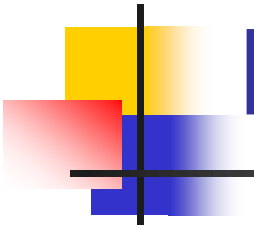
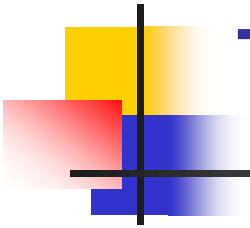


CSIT 315
SE 1
Top-Level Design



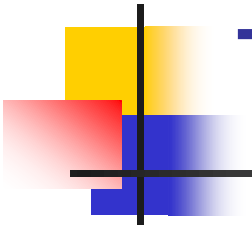
Design and Quality

- Characteristics of a good design:
  - Implements all explicit requirements; and those desired by the customer
  - Is a readable, understandable guide for the programmers, testers and those who will support the software
  - Provides a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective



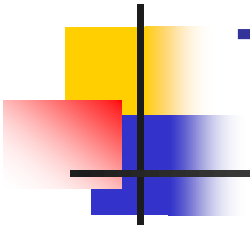
Technical Criteria

- Design should exhibit an architectural structure that:
  1. Has been created using recognizable design patterns
  2. Is composed of components that exhibit good design characteristics
  3. Can be implemented in an evolutionary manner



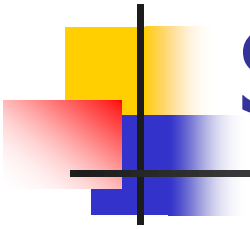
Technical Criteria

- Design should be modular.
- Should contain distinct representations of data, architecture, interfaces, and components (modules)
- Should lead to data structures that are appropriate for the objects to be implemented



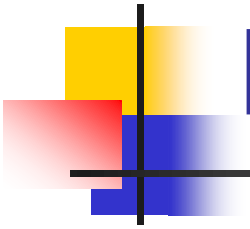
Technical Criteria

- Design should:
  - lead to components that exhibit independent functional characteristics
  - Lead to interfaces that reduce the complexity of connections between modules and with the external environment
  - Be derived using a repeatable method that is driven by information obtained during requirements analysis



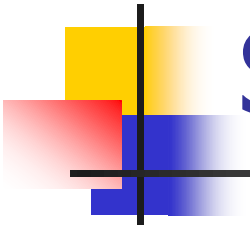
Software Design

Remember Design Principles??



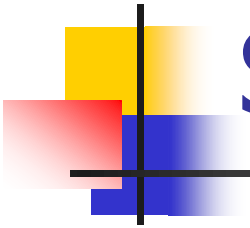
Reminder!!!

- Next deliverable:
  1. Test suite
     - … and not long after!!!
  2. Top-level design
- Remember!!!
  - Team-Work!
    - Division of Labor



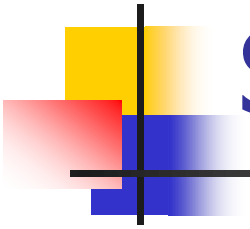
Software Design

- Recall
  - System design concentrate on the modules in the system and how they interact with each other
  - Spec of a module is often communicated by its name $\rightarrow$ communicate its functionality
  - Spec of a module are conveyed by our understanding of phrases that label the module
  - In design, a more detailed specification is given
    - Explain in natural language what a module is supposed to do



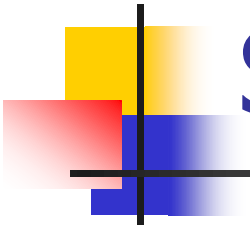
Software Design

- Correct implementation of the module depends on the coder's interpretation of the modules
- Purpose of design is to plan a solution of the problem specified in the spec document
- Perhaps the most crucial factor affecting the quality of the software
- Has a major impact on the latter phases, particularly testing and maintenance
- Output of this phase is the design document – similar to a blueprint or plan for the solution



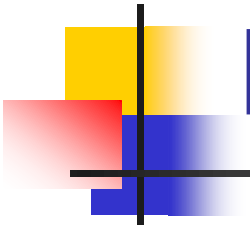
Software Design

- Two levels of design:
  1. Top-level (a.k.a system design)
  2. Detailed
- Top-level identifies:
  - modules in the system
  - the specs of these modules
  - How the modules interact with each other to produce the desired results



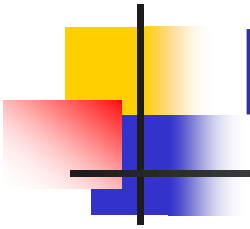
Software Design

- At end of system design:
  - All major data structures, file formats, output formats, etc are determined
  - Internal logic of each module specified in the specification is decided – in natural language (a.k.a. pseudocode)



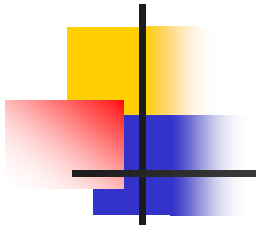
Design (Levels of)

- Top-Level:
  - Pseudo-code!!!!
  - Specific algorithms are selected
  - Data structures are chosen
- Detailed design
  - Each module is refined in detail
- From an abstraction viewpoint, during this activity, the fact that the modules are to be interconnected to form a complete product is ignored
- A program, but NOT SPECIFIC CODE



Detailed Design

- The focus is on refining, in detail, the logic specified in the top level.
  - All details of each module are refined
    - Module names and parameters
    - Calls, etc



- Specify modules in a descriptive language independent of the target language
- The result of this phase is the program/code, except it is not written in any programming language
- The benefit of this?

High-Level Requirements for Top-Level Design

- Document
  - acceptance of guiding principals
  - operational concepts
  - institutional framework, and design principles
- Highlight any exceptions and provide rationale for this exception
- focusing on unique state circumstances, impact (or lack of impact)

SYSTEM DESIGN

- Present the proposed system design for deployment.
- Define the interfaces required between/among systems and the interface documents.
  - This section is expected to be approximately 15 pages in length.]

Architecture Overview

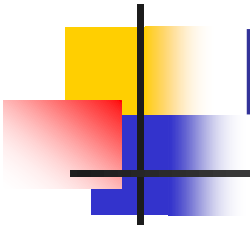
- Provide an overview of the architecture.
- It should summarize the key concepts (e.g., single sign-on for enforcement officers to access any info they need) that shape the design.
- It should summarize key aspects of the approach chosen to implement the system(e.g., Web services). xplain
-

Architecture Overview (2)

- It should include the System Design Diagram (and Network diagram, which highlight new and modified systems and networks. .
- **Note: All the names used on the System Design Diagram should also be found on the Network Diagram, and they should be consistent.**

Architecture Overview (3)

- If the design proposed is not represented in or aligned with the Architecture explain how and why.
- Include sufficient detail to explain all departures from the standard architecture.



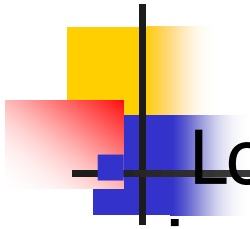
Project Design Elements

- This section should include a subsection for each of the module.
- For each module, a table showing the interface requirements (existing and planned) with other systems and the interface types that will be employed (where known) should be included.

Top-Level Sample

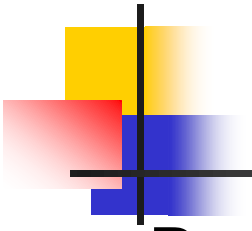
```
proc get_top_level_instances_matching { wildcard } {
# Make a variable to hold the top-level instances that match the wildcard    catch { array unset
    names_to_return }
   array set names_to_return [list]

# The collection of names is all the hierarchies in the design
   for each_in_collection name_id [get_names -filter * -node_type hierarchy] {
# The short_full_path option gets the name in the form
# instance|instance|...
# It uses only instances regardless of whether the
# "Display entity name for node name" setting is on or off
   set short_full_name [get_name_info -info short_full_path $name_id]
# Split the hierarchy into a list, breaking it apart on the
# hierarchy separator |
   set short_full_pieces [split $short_full_name "|"]
```

Top-Down Design



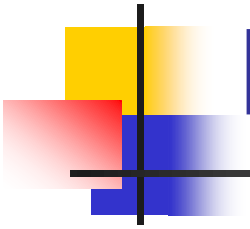
- Looking at a problem as a whole, it may seem impossible to solve because it is so complex. Examples:
  - writing a tax computation program
  - writing a word processor
- Complex problems can be solved using **top-down design**, also known as s**tepwise refinement**, where
  - We break the problem into parts
  - Then break the parts into parts
  - Soon, each of the parts will be easy to do

Advantages of Top-Down Design



- Breaking the problem into parts helps us to clarify what needs to be done.
- At each step of refinement, the new parts become less complicated and, therefore, easier to figure out.
- Parts of the solution may turn out to be reusable.
- Breaking the problem into parts allows more than one person to work on the solution.

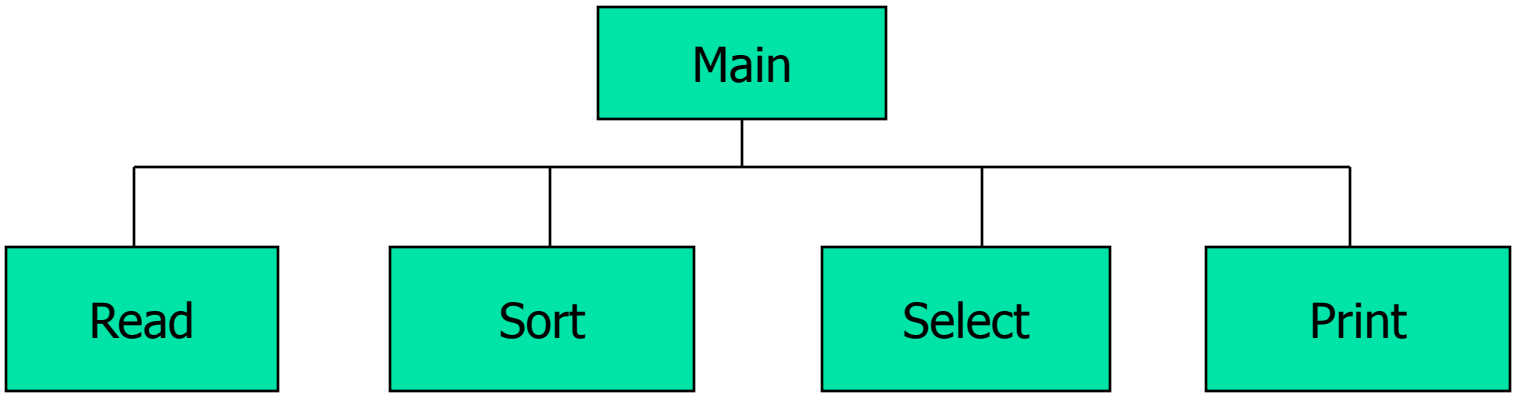
An Example of Top-Down Design



- Problem:
  - We own a home improvement company.
  - We do painting, roofing, and basement waterproofing.
  - A section of town has recently flooded (zip code 07043).
  - We want to send out pamphlets to our customers in that area.

The Top Level

- Get the customer list from a file.
- Sort the list according to zip code.
- Make a new file of only the customers with the zip code 07043 from the sorted customer list.
- Print an envelope for each of these customers.



Another Level?

- Can any of these steps be broken down further? Possibly.
- How do we know? Ask yourself whether or not you could easily write the algorithm for the step. If not, break it down again.
- When you are comfortable with the breakdown, write the pseudo-code (top-level design) for each of the steps (**modules**) in the **hierarchy**.
- Typically, each module will be coded as a separate function.

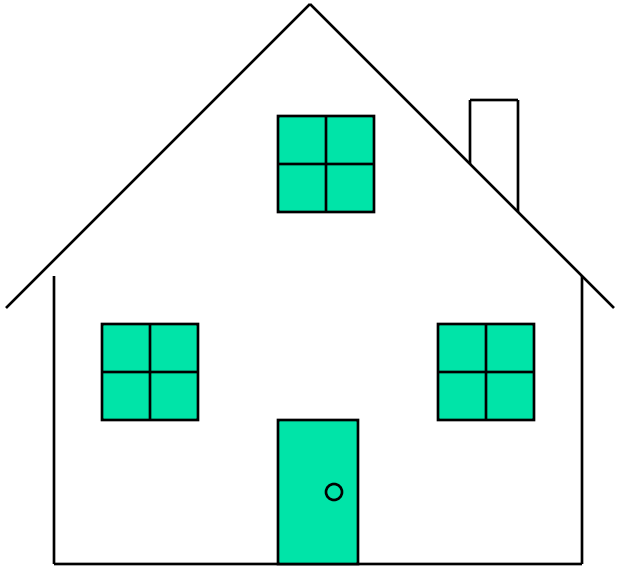
Structured Programs

Using top-down and Top-level design for all programming projects is recommended. Why?

- The standard way of (writing programs) developing systems.
- Systems produced using this method and using the three kinds of control structures, sequential, selection and repetition, are called **structured programs**.
- Structured programs are easier to test, modify, and are also easier for other programmers to understand.

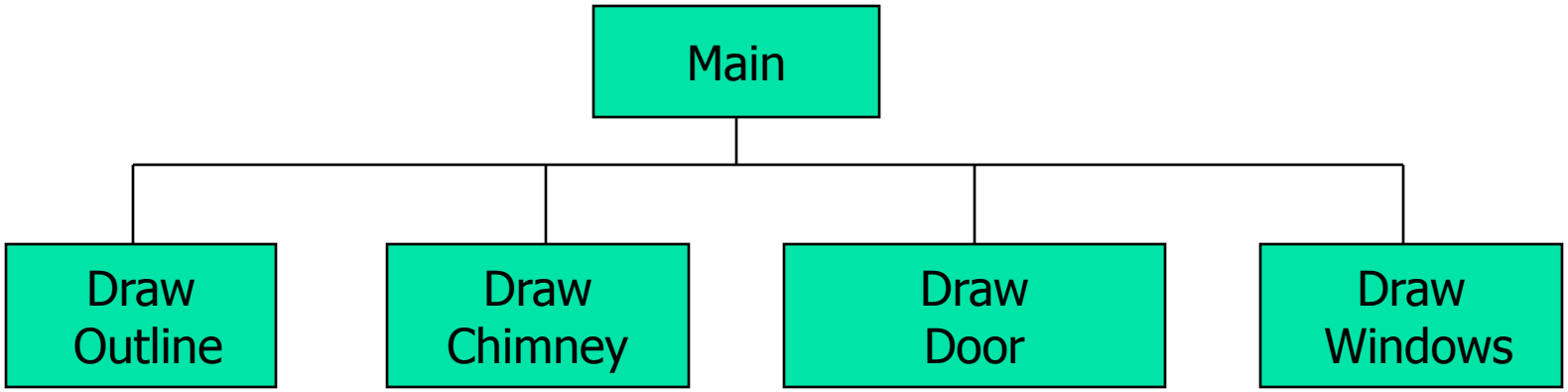
Another Example

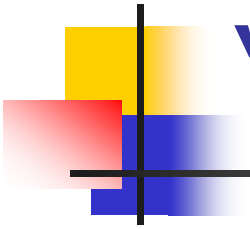
- Problem: Write a program that draws this picture of a house.



The Top Level

- Draw the outline of the house
- Draw the chimney
- Draw the door
- Draw the windows





“Top-level” Design for Main

Call Draw Outline

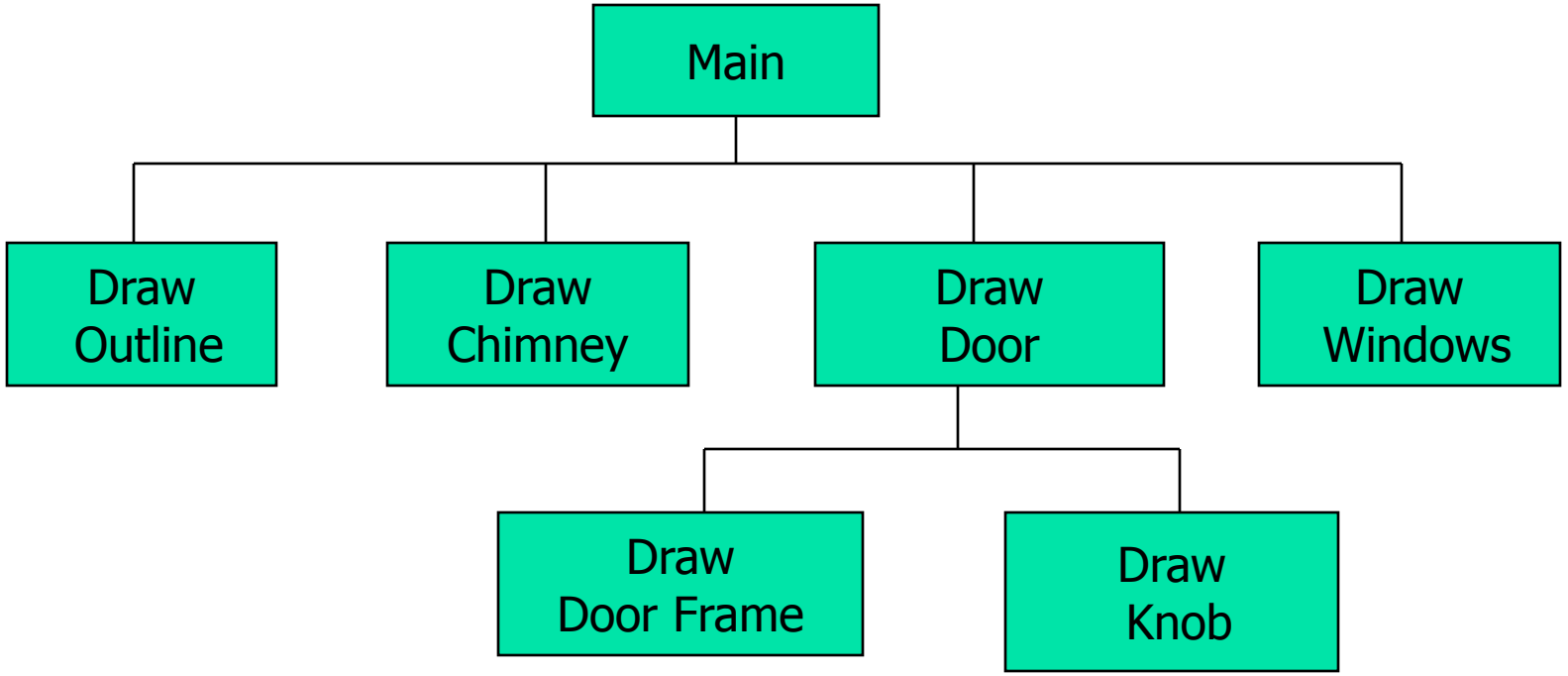
Call Draw Chimney

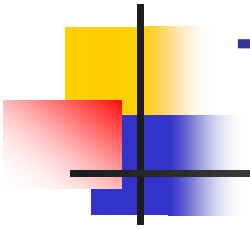
Call Draw Door

Call Draw Windows

Observation

- The door has both a frame and knob. We could break this into two steps.





Top-Level Design for Draw Door

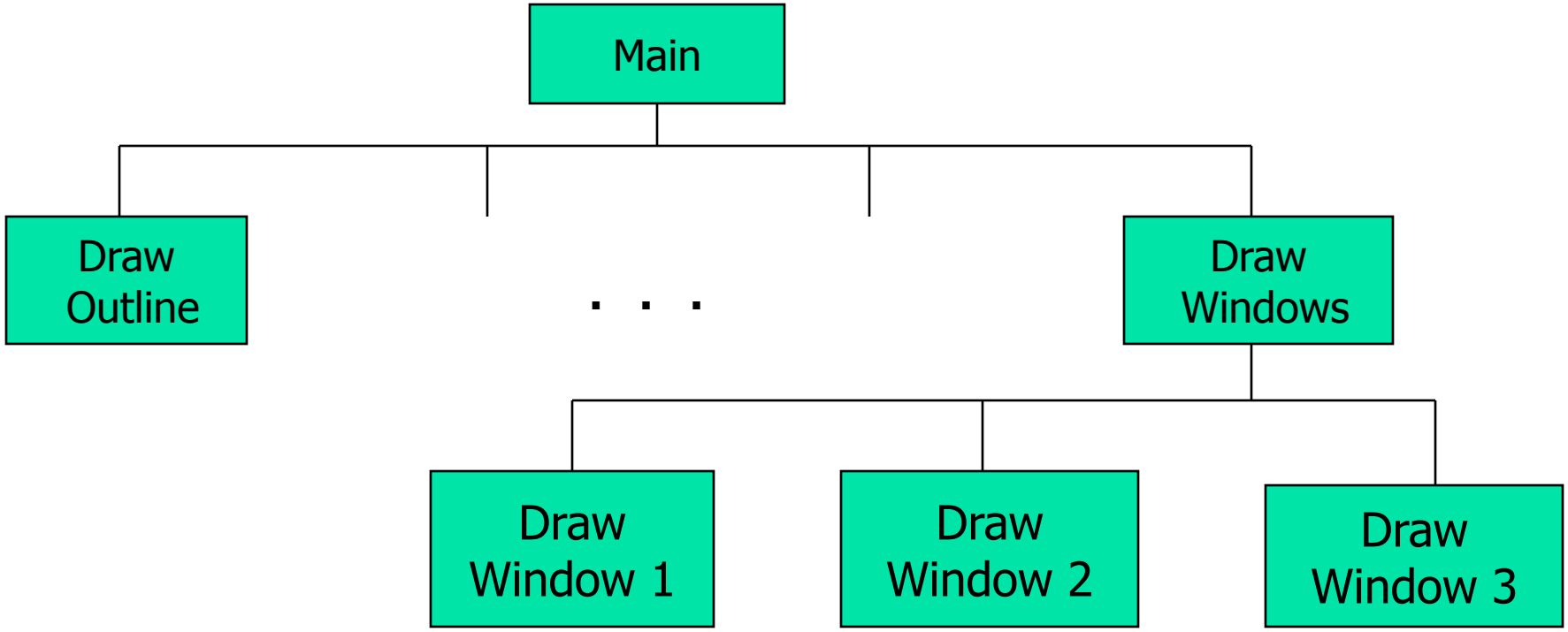
Call Draw Door Frame

Call Draw Knob

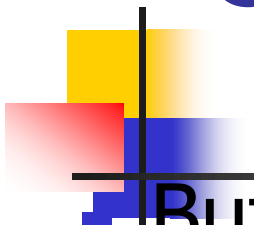
Note: More of the module "logic" needs to be added.

Another Observation

- There are three windows to be drawn.

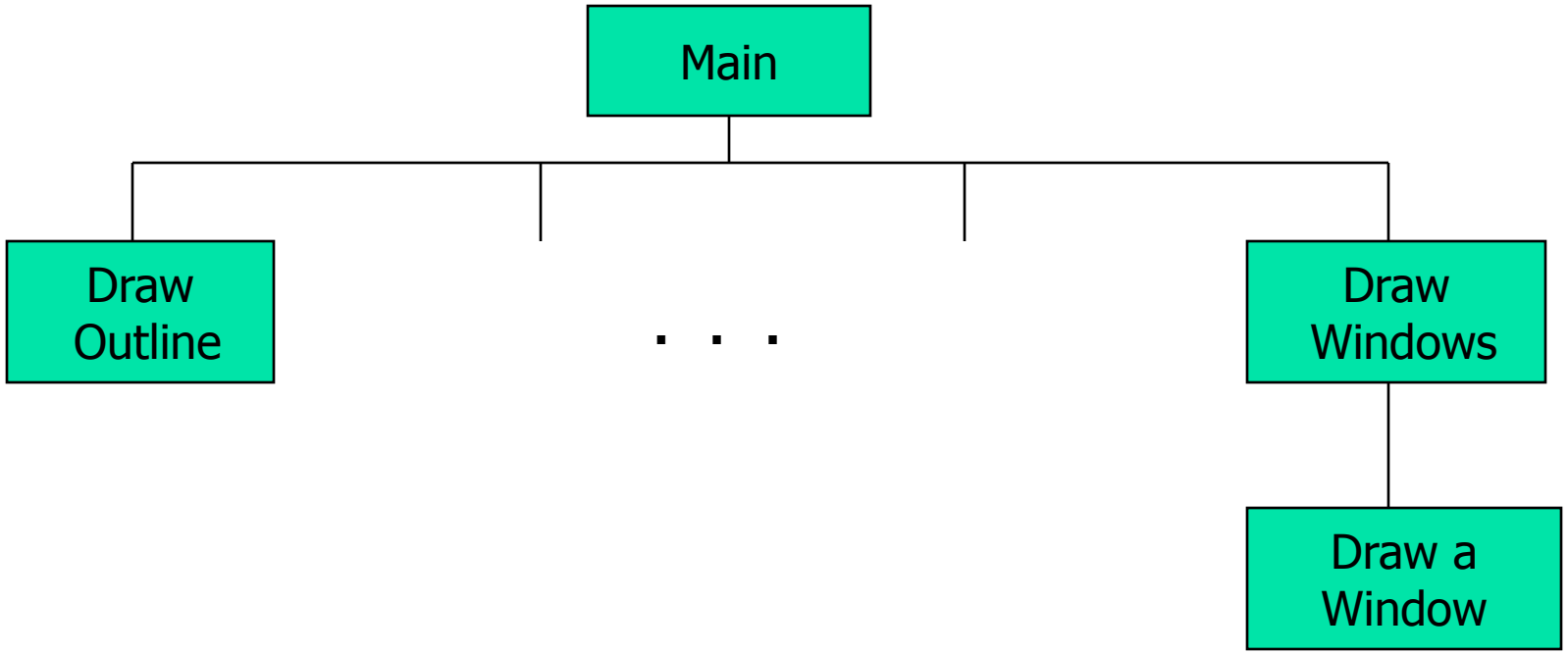


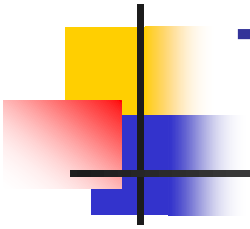
One Last Observation



- But don't the windows look the same? They just have different locations.
- So, we can reuse the code that draws a window.
  - Just copy the code as many times as needed and edit it to place the window in the correct location, or
  - Use the code three times, "sending it" the correct location each time (we will see how to do this later).
- This is an example of **code reuse**.

Reusing the Window Code





Top-Level for Draw Windows

Call Draw a Window, sending in Location 1

Call Draw a Window, sending in Location 2

Call Draw a Window, sending in Location 3

Again!! The logic involved in each of these would need to be given