# Getting control of exceptions

*Presented to the Workshop on Exception Handling in Object Oriented Systems: towards Emerging Application Areas and New Programming Paradigms*

21 July 2003

William Bail

The MITRE Corporation and The University of Maryland

# Overview

- Our being able to control exceptions has provided a significant advantage in being able to write more reliable software
- While not explicitly helping us avoid errors, they enable us to detect their presence and control their effects
- Yet they act in opposition to much of what we have learned is good software design – simple structures with well-defined control flows
- In addition, they complicate the process of performing formal analyses on our systems
- In this talk we explore these issues

Extension of work reported in W.G. Bail. “Exception-handling Design Patterns”. *Advances in Computers*, vol. 49. Academic Press, 191-238, 1999

# Outline

- History of exceptions – a love/hate relationship
  - What are they?
  - Why do we care?
  - How do we cope with them?
- The challenges
  - Complexity
  - Program proofs of correctness
  - Object-orientation
  - Others
    - Component-based design (and COTS)
    - Concurrency

# Our relationship with exceptions

- Exceptions have been with us for a long time
  - Originally uncontrolled – detected by entities outside of our software programs
  - e.g., Grace Hopper's moth – unannounced
- May occur when execution enters an error state
  - Often caused by code defects
  - or hardware (moth)
  - or erroneous inputs (wrong key)
- Mostly unwelcome

  ABEND

- Luckily, software rarely has defects...

# What are “exceptions”, really?

- A condition dealing with the interaction between a *client* and a *server (a customer* and a *provider)*
  - Client – the user of a capability
  - Server – the provider of a capability
  - Capability – some service (+, /, alloc(), SORT, Kalman filter,...)
- *Exception* – the unexpected or unusual condition where the server fails to carry out a requested or expected operation
  - State of system is considered to be “abnormal” (error state)
  - Server may attempt recovery (*handle exception*) or inform client that the operation failed (*propagate exception*)

# What causes abnormal states?

- Two general sources:
  - Software defects – erroneous logic in the code - caused by
    - mistakes made by programmers
    - mistakes in the design or in the requirements
    - erroneous code produced by a compiler or application generator
    - caused by hardware failure (processor fault)
  - Erroneous data – improper or unexpected data – caused by
    - input by an operator
    - transmitted by another system
    - be corrupted by faulty hardware or by a noisy communications channel (moth)
    - exist in databases used by the software

# Options for coping with exceptions

- Just let them happen (historical precedent), or
- Try to control them via exception handling (*i.e.*, a class of mechanisms and design approaches for dealing with effects of abnormal (exceptional) conditions), such as:
  - Correct the problem and retry (*forward recovery*)
  - Roll-back to a previous state and continue (*backward recovery*)
  - Attempt correction and continue at a reduced capability (*graceful degradation*)
  - Skip the operation and continue
  - Terminate execution under control (*fail-safe*)
  - Propagate the exception (let someone else worry about it)
  - Record the exception to a logging file

# Handling exceptions

- Exception handling involves three distinct functions:
  - Detection – determining that the current state of computation is abnormal (of course, we have to know what is abnormal)
  - Propagation – notifying client that an error state has been detected (in a way that allows them to take action)
  - Handling – performing recovery actions to mitigate the effect(s) of the error state
- Programming languages may provide mechanisms for working with exceptions
  - Allowing us to separate exception handling from normal processing, reducing design complexity
- If not, we need to insert exception handling code manually (defensive programming)

# Maturation of exception handling

- Originally, we just corrected the code
- Later, after getting tired of their sneaking up on us, we started to watch for them and control them by inserting defensive code
- Still later, after getting tired of writing all that extra code, we added capabilities into languages to help us
  - Middle 1960s – PL/1
  - Followed by many other languages, incl CLU, Ada83, C++, Ada95, Eiffel, Java, ...
- We have gotten much better
  - but each new SW technique (*e.g.*, OO) brings new challenges
  - Exceptions just don't fit in – they disturb our nice abstractions

# Why do we want to handle exceptions?

- Random failures get annoying
  - Especially when someone else tells us about them (e.g., the OS)
- They
  - remind us of our failures
  - interrupt us when we are in the middle of doing real work
  - perturb our natural flow of logic
  - make us think defensively
- But, if there is always a chance of failure, but never knowing when, our systems are not dependable
  - We never know what is going to happen
  - Not a good thing in general
  - How does your plane's software handle exceptions? Ignore them?

# Strategies

- Two general strategies: show that exceptions won't happen, or control them when they do
- Show that they won't happen (before execution)
  - Restrict language use
    - e.g., SPARK / Ravenscar
  - Demonstrate that error states cannot happen via logic
    - Made possible by restricted logic
    - (of course, random cosmic waves or moths could always occur)
- Control them when they do (after execution)
  - Extensive use of run-time checks (explicit)
  - Establish system-wide error handling policies as a part of design
    - and follow them

# Aspects of error handling policies

- Four parts to design process:
  - Allocation of responsibility – the assignment of error handling responsibility in the detection, propagation, and correction of error states (exception-handling protocols)
    - *Who does what?*
  - Exception semantics – the conditions which the exceptions are to signal
    - *What is it telling me?*
  - Global design patterns – the overall structure of exceptions across the system to be developed
    - *How does it all work together?*
  - Local design patterns – (implementation mechanisms) – the specific language features that will be used to implement the exception handling logic
    - *How do I do it?*

# The challenges

- Complexity
- Program proofs of correctness
- Object-orientation
- Component-based design (and COTS)
- Concurrency

# Challenges – effect on complexity

- By adding new (and hidden) control paths, exceptions make our software more complex
- Consider effect on cyclomatic complexity
  - Explicit error handling adds execution paths – hence increases complexity
  - Implicit error handling...the same effect...but not as obvious
- Sample function – a binary search written in Ada83
- *Remember, cyclomatic complexity V(g) = number of independent paths through code segment*

# Sample function

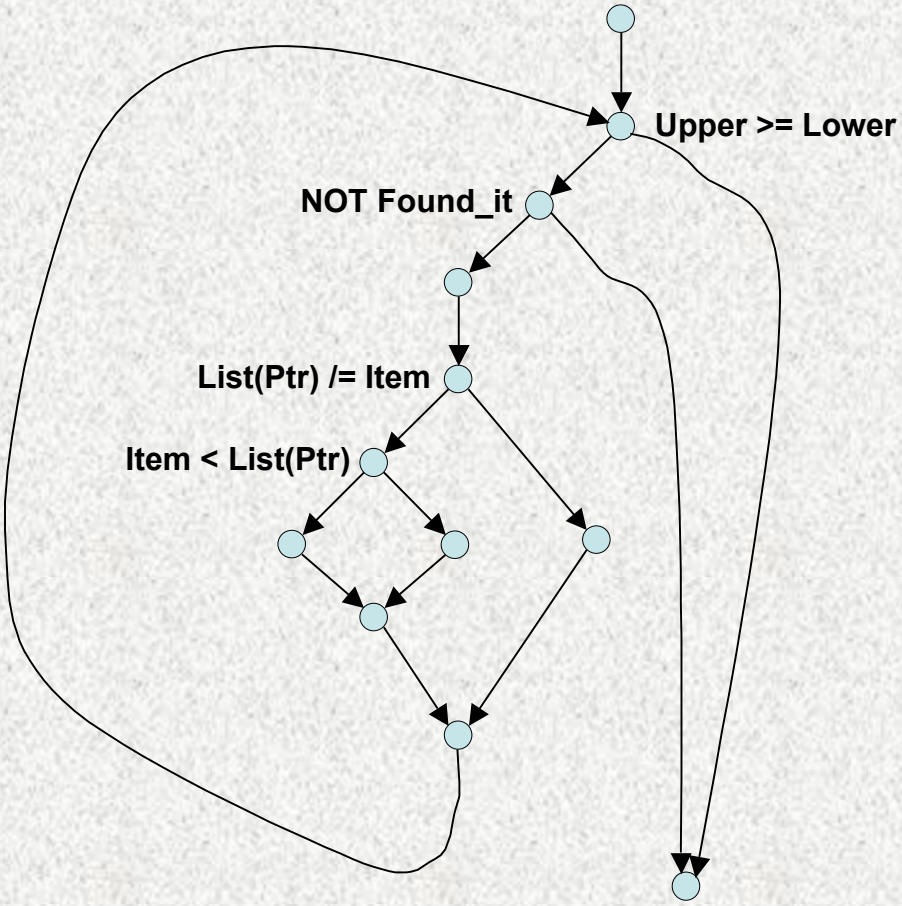
## A simple binary search

```
FUNCTION Search (
        Item: Integer;
        List: ItemList;
        Start_Pos: Integer;
        End_Pos: Integer)
        RETURN Integer IS
    Result: Integer ;
    Upper, Lower, Ptr: Integer RANGE
                            List'RANGE;
    Found_it : Boolean;
BEGIN
    Lower := Start_Pos;
    Upper := End_Pos;
    Result := 0;
    Found_it := False;
    WHILE ( (Upper >= Lower) AND
        (NOT Found_it )) LOOP
        Ptr := (Upper+Lower)/2;
        IF ( List(Ptr) /= Item ) THEN
            IF ( Item < List(Ptr) ) THEN
                Upper := Ptr - 1;
            ELSE
                Lower := Ptr + 1;
            END IF;
        ELSE
            Found_it := True;
            Result := Ptr;
        END IF;
    END LOOP;
    RETURN Result;
    END;
```

# Preconditions for the code

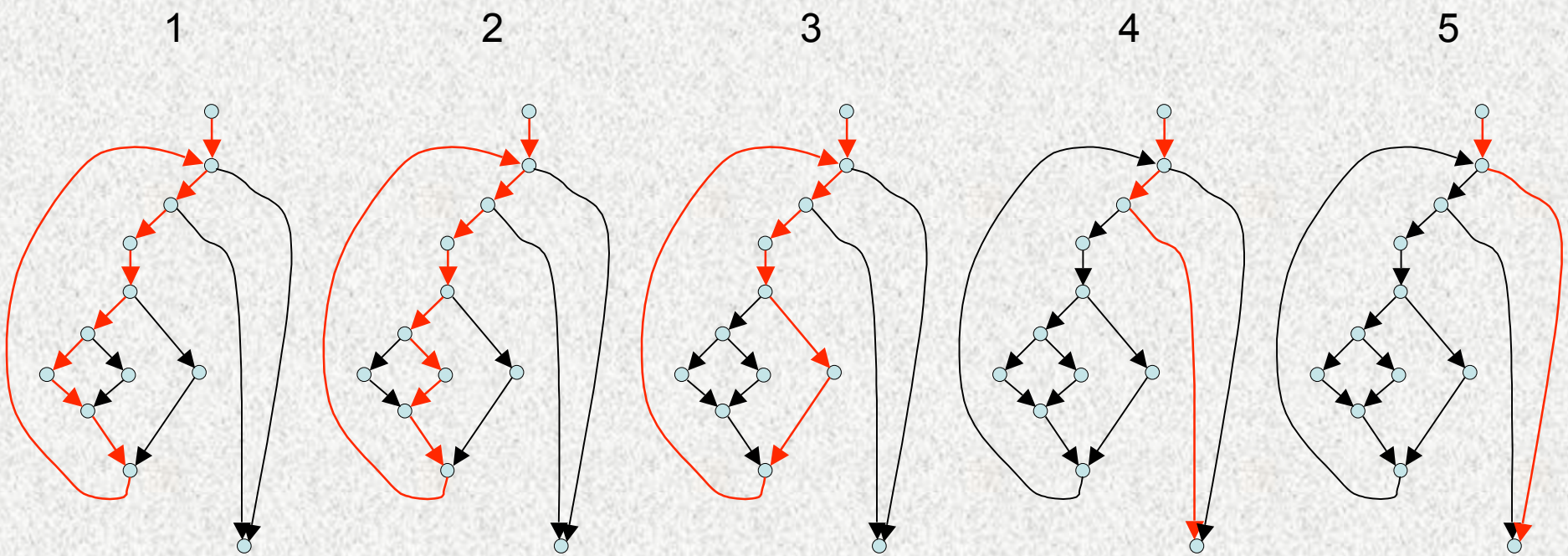
- For the code to execute correctly, certain conditions must be met:
  - The subrange provided is within the range of the array
  - The start of the subrange is less than the end of the subrange
  - The list is sorted in ascending order
- However, no explicit checks are performed in the code to verify that these assumptions are valid
- Let's compute V(g)

# Function flow chart




Rather simple
Fairly clean

# Independent paths




Five independent paths, so V(g) = 5

# But wait...

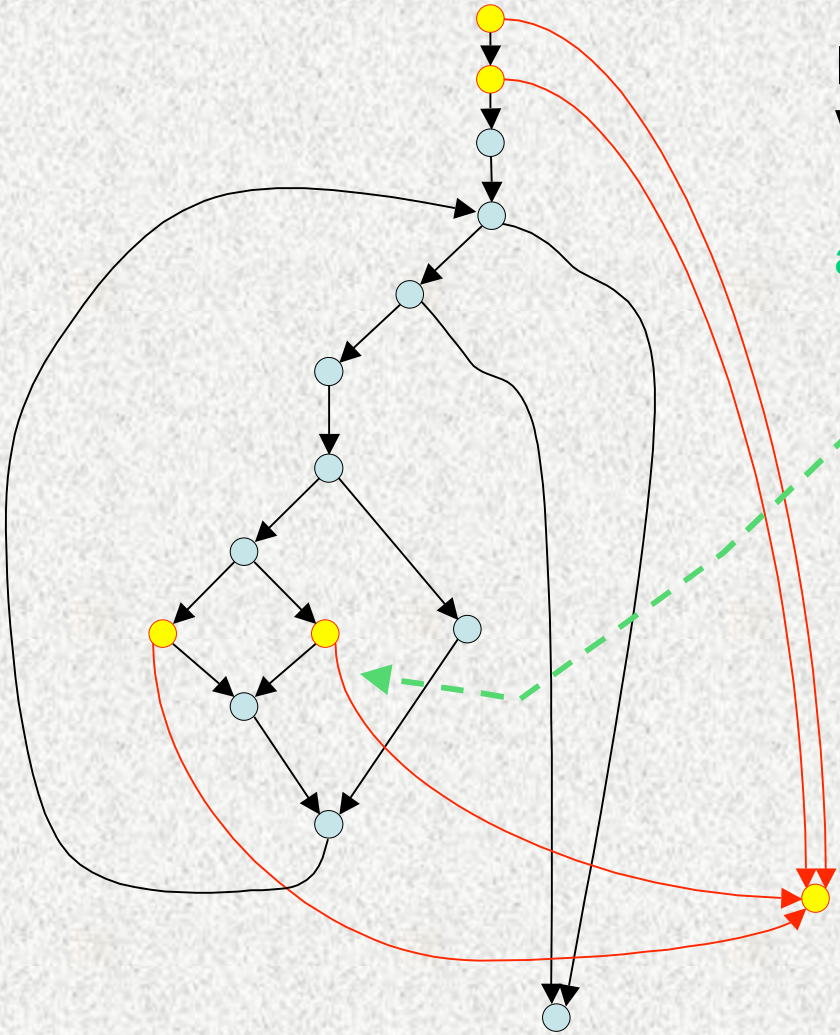
- What happens if we consider implicit exception handling?
  - According to cyclomatic complexity rules, we need to include all paths
- We need to analyze the code to determine what the implicit paths are
- Since this was written in Ada, such paths exist

# Analysis of code

```
FUNCTION Search ( Item: Integer;
        List: ItemList;
        Start_Pos: Integer;
        End_Pos: Integer)
        RETURN Integer IS
        Result: Integer ;
        Upper, Lower, Ptr: Integer RANGE List'RANGE;
        Found_it : Boolean;
BEGIN
        Lower := Start_Pos;
        Upper := End_Pos;
        Result := 0;
        Found_it := False;
        WHILE ( (Upper >= Lower) AND
                (NOT Found_it )) LOOP
                Ptr := (Upper+Lower)/2;
                IF ( List(Ptr) /= Item ) THEN
                        IF ( Item < List(Ptr) ) THEN
                                Upper := Ptr - 1;
                        ELSE
                                Lower := Ptr + 1;
                        END IF;
                ELSE
                        Found_it := True;
                        Result := Ptr;
                END IF;
        END LOOP;
        RETURN Result;
        END;
```

Potential for implicit execution path due to exceptions

# Results in new control flow graph



**No longer simple and clean**
**V(g) = 9 (almost twice as complex)**

**and this is really annoying**

**exception return**

```
FUNCTION Search ( Item: Integer;
        List: ItemList;
        Start_Pos: Integer;
        End_Pos: Integer)
        RETURN Integer IS
        Result: Integer ;
        Upper, Lower, Ptr: Integer RANGE List'RANGE;
        Found_it : Boolean;
BEGIN
        Lower := Start_Pos;
        Upper := End_Pos;
        Result := 0;
        Found_it := False;
        WHILE ( (Upper >= Lower) AND
                (NOT Found_it )) LOOP
                Ptr := (Upper+Lower)/2;
                IF ( List(Ptr) /= Item ) THEN
                        IF ( Item < List(Ptr) ) THEN
                                Upper := Ptr - 1;
                        ELSE
                                Lower := Ptr + 1;
                        END IF;
                ELSE
                        Found_it := True;
                        Result := Ptr;
                END IF;
        END LOOP;
        RETURN Result;
        END;
```

# More bad news

- It is even worse than that
- In order to compute V(g) need to analyze environment of function for other declarations
- Additional exceptions may be possible
  - or the ones identified may be not possible
- This obscures the real structure of the code
  - Making it harder to think and reason about it
- Major effects
  - Perturbs our normal thought process
  - Execution no longer smooth and continuous
  - Requires complicated analysis process

# Challenges – effect on formal methods

- Remember
  - Structured programming?
  - Go to statement considered harmful
- Belief was (and still is) that certain program constructs promoted understanding and the ability to generate proofs of program correctness
- Harlan Mills limited developers to 7 constructs:

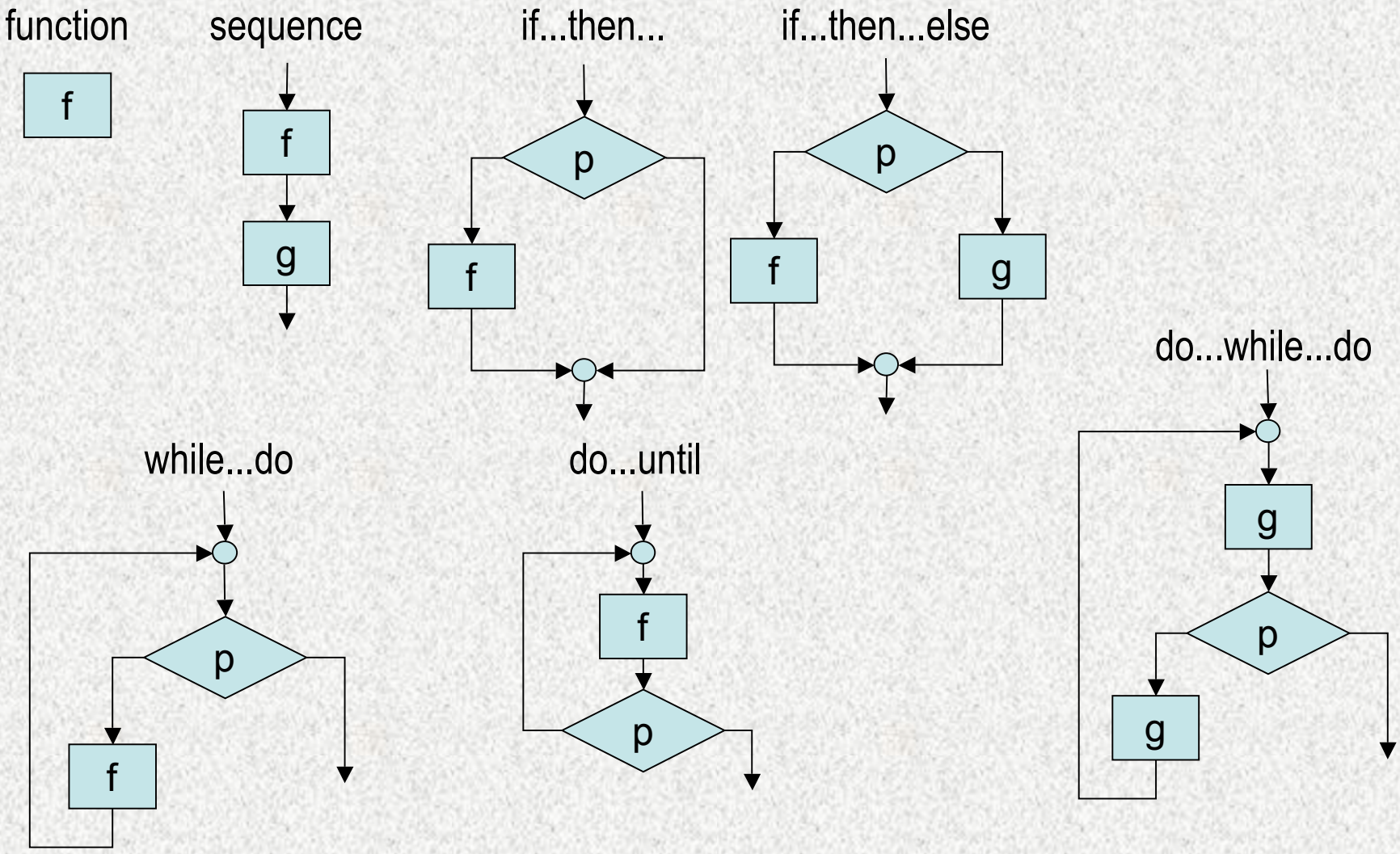
**Function** – f
**Sequence** – f; g
**If…then** – if p then f
**If…then…else** – if p then f else g;

**Whiledo** – while p do f ;
**Do until** – do f until p;
**Do…while…do** – do f while p do g ;

# Why are these important?

- These forms have well defined semantics that support formal verification
  - and informal understanding
- These forms are prime programs
  - A *prime* program is a proper program that has no proper subprogram of more than one node
- are used to form proper programs
  - A *proper* program
    - has a single entrance and a single exit, and
    - all nodes have a path through that node from the entry to the exit

# What they look like...



# So what?

- For each proper program, its computation can be expressed in a logical form
  - Made possible by the single entry and single exit point (reduce side effects)

the function

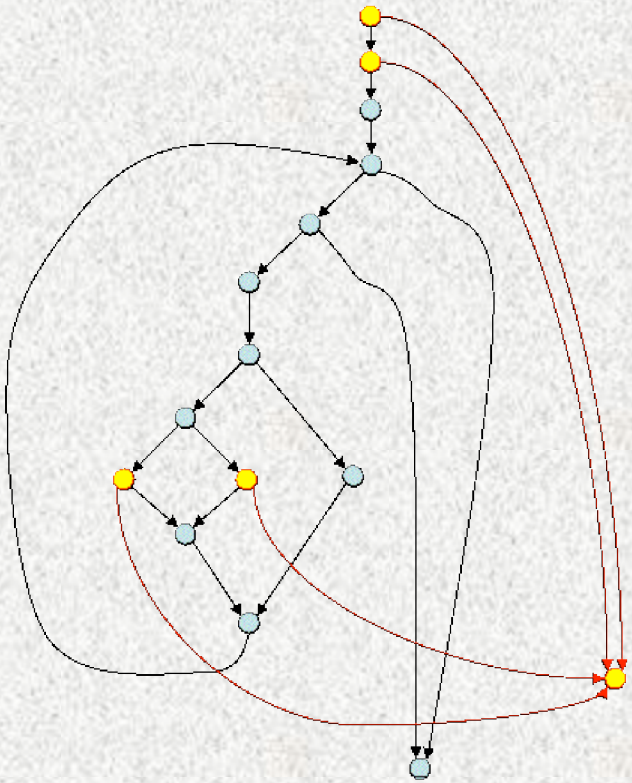
```
f:
if
    p
then
    g
end if
```

can be expressed as

$(p \Rightarrow f = g \mid \neg p \Rightarrow nil)$

# But what about...

Where is the proper program? and what is its function?

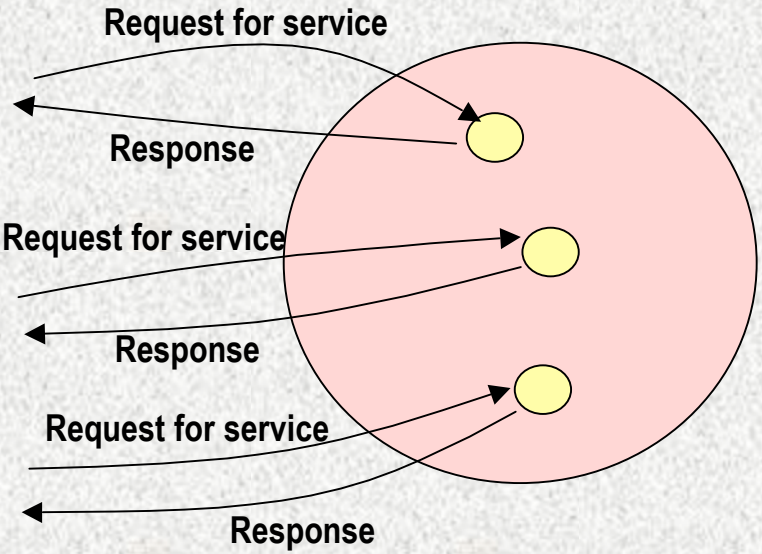


# Challenges – exception handling in OO

- Current approach – method-based exception Propagation
  - Granularity increments between method and client
- Recommended approach – object-based exception propagation
  - Added object level of granularity

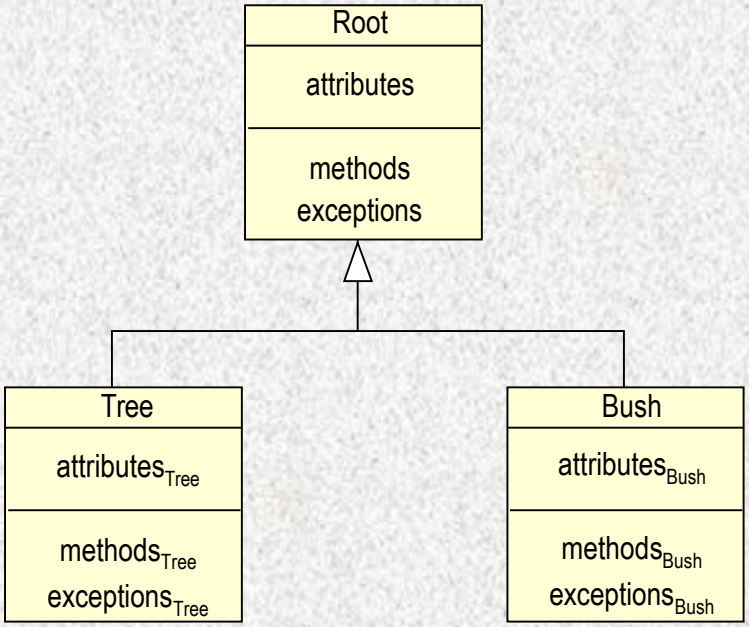
# Method-based exception propagation

- Typical approach
- Granularity at method level
  - Function-based concept
  - Direct client to server interaction
- Loss of control of grain size
  - No direct filtering at object level
- Fails to provide object with appropriate level of authority and responsibility



# Inheritance and object-based exceptions

- Through inheritance, specialized subclasses receive class-wide exceptions and handlers
- Not method based – hence can exploit common handling strategies
- Able to update class-wide handlers without disturbing existing ones



# Challenge – component-based design

- Constructing systems out of pre-existing components is seen as a viable, cost-effective strategy for future system development
- Perhaps by using a module interface language (MIL)
- But...
- Unless the components adhere to a common error handling policy, disjoint handling of exceptions will occur
  - e.g., what about COTS?
- Requires establishing domain-wide architectural requirements
  - or creation of standard wrappers around non-conforming components...but how?

# Challenge – concurrency

- Time-based logic is inherently very difficult
  - Needed to understand behavior of real-time systems
- Based on analysis of time demands and logic of design/code
- Exception handling places new and additional challenges to the analysis of this logic
  - *"Where did that come from?"*
- Need to understand the underlying principles behind how to select architectural features
  - What should we do, what shouldn't we do, how do we know when and where?

# Parting message

- We have come far
- We have exceptions partially under control
- But where we want to go has additional challenges

# References


- Amey, Peter, and Roderick Chapman. Industrial Strength Exception Freedom. ACM SIGAda Annual International Conference (SIGAda 2002)
- Bail, W.G. “Exception-handling Design Patterns”. *Advances in Computers*, vol. 49. Academic Press, 191-238, 1999
- Goodenough, J. (1975) “Exception Handling: Issues and a Proposed Notation”, *Communications of the ACM* (December 1975), 683-696
- Linger, R.C, H.D. Mills, and B. I. Witt. *Structured Programming – Theory and Practice.* Addison-Wesley: Reading MA. 1979.