Models for SwQA

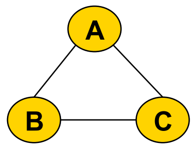
From S. Somé, A. Williams

Models for assessing software quality

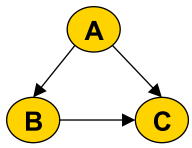
- Models are a standard engineering practice for performing analysis of a system.
- Properties of a good model:
  - Simpler than the actual system, but preserves relevant attributes of the system.
  - Compact: small enough to be comprehensible – either for human or machine processing, and to be created with less effort than an actual system.
  - Predictive: The model must represent a relevant characteristic well enough to distinguish between good and bad outcomes of an analysis.
  - Semantically meaningful: a failure diagnosis with respect to a model should be equally applicable to the actual system.
  - Sufficiently general: Models intended for use over a range of systems should be applicable across the domain.

Graph representations of software (1)

- Graph: set of nodes, and a set of edges between nodes.
  - Nodes: {A, B, C}
  - Edges: { (A,B), (B,C), (C,A) }



- Directed graph: “arcs” replace “edges”, and the order in the pair becomes significant.

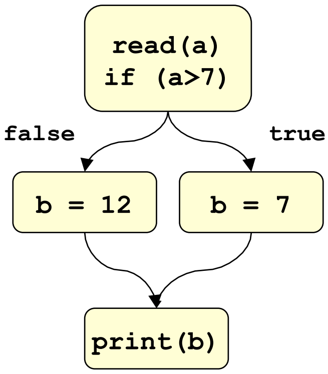


Graph representations of software (2)

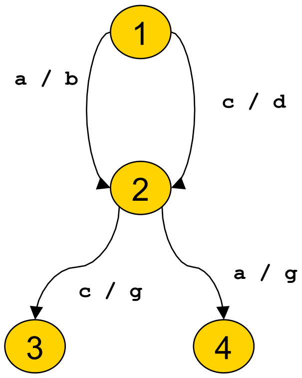
- Software execution can be considered as a sequence of states alternated with actions (i.e. machine operations) that modify the system state.
- A graph can be used as the execution model in two ways:
  - Option 1: Actions occur within graph nodes, and an edge represents a system state while control flow transfer occurs.
  - Option 2: A system state is represented by a graph node, and actions occur on the graph edges.

Examples of graph models

- Option 1: A control flow graph.



- Option 2: A finite state machine.



Notation: **input / output**

Scenario Graph

Scenario Graph

- Generated from a use case
- Nodes correspond to point where system waits for an event
  - environment event, system reaction
- There is a single starting node
- End of use case is finish node
- Edges correspond to event occurrences
  - May include conditions and looping edges
- Scenario:
  - Path from starting node to a finish node

Use Case Scenario Graph (1)

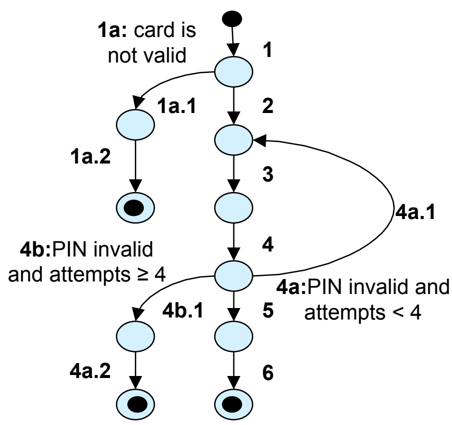
Title: User login

Actors: User

Precondition: System is ON

1. User inserts a card
2. System asks for personal identification number (PIN)
3. User types PIN
4. System validates user identification
5. System displays a welcome message to user
6. System ejects card

Postcondition: User is logged in



Use Case Scenario Graph (2)

Alternatives:

1a: Card is not valid

1a.1: System emits alarm

1a.2: System ejects card

4a: User identification is invalid

AND number of attempts < 4

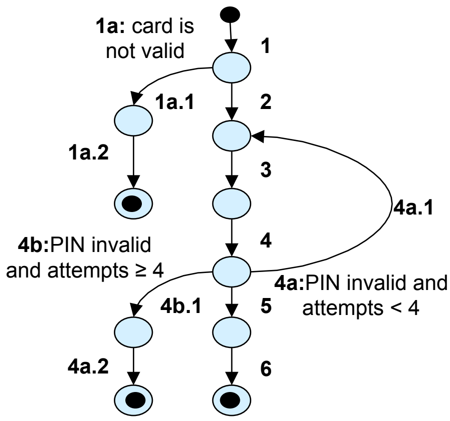
4a.1 Ask for PIN again and go back

4b: User identification is invalid

AND number of attempts ≥ 4

4b.1: System emits alarm

4b.2: System ejects card



Control Flow

Control flow graphs

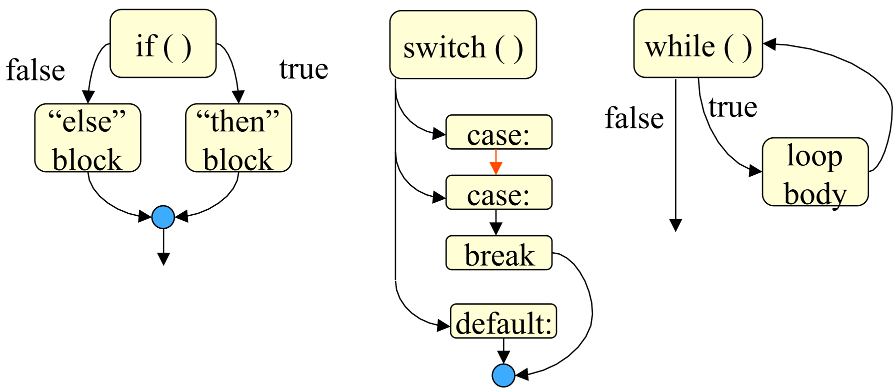
- Intraprocedural flow graph:
  - Models the internal paths of control flow within a single procedure or method.
  - Graph nodes represent a code block.
  - Graph arcs represent alternative paths to what code statements might be executed next.
- Interprocedural flow graph, or call graph:
  - Models the potential sequences of calls to various methods.
  - Graph nodes represent methods.
  - Graph arcs indicate that the method at the head of the arc can call the method at the tail of the arc.
  - Polymorphism makes such graphs more complex…

Intraprocedural flow graphs

- What is included within a single node:
  - A single-entry, single-exit series of statements.
    - If there are several entry points to a code statement, the statement must be the first line in a node.
  - Assumption: if you enter the node, all statements in the node will be executed.
    - What about exceptions?
- Arcs (or edges) should be used whenever there is an alternative as to what could be executed next, or to transfer to a statement for which there are multiple entry points.
  - It should be clear from the diagram as to the conditions required to select one alternative over the others.

Intraprocedural flow graphs

- Based on programming language constructs.

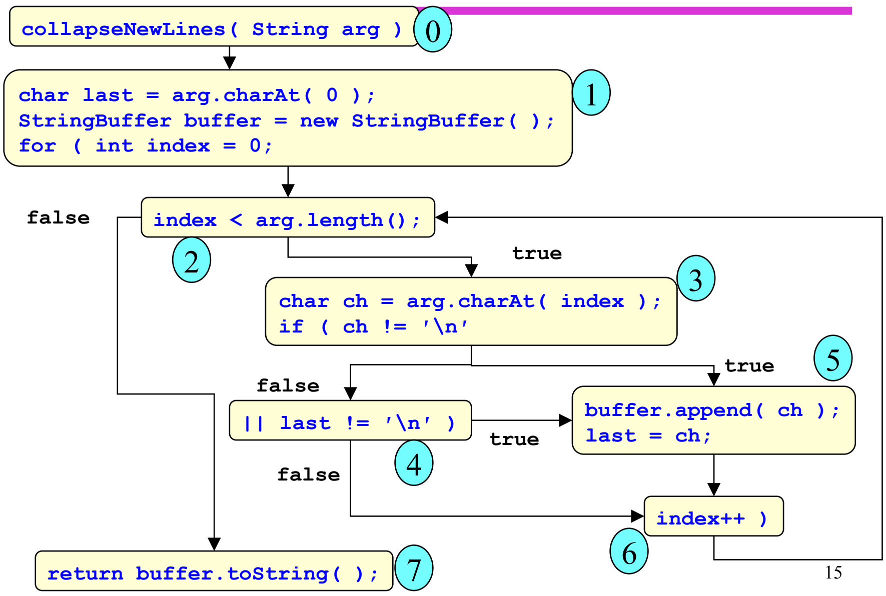


Java Example

- Function: replace n ≥ 2 consecutive new line characters with a single character.

```
public static String collapseNewLines( String arg )
{
   char last = arg.charAt( 0 );
   StringBuffer buffer = new StringBuffer( );
   for ( int index = 0; index < arg.length(); index++ )
   {
      char ch = arg.charAt( index );
      if ( ch != '\n' || last != '\n' )
      {
         buffer.append( ch );
         last = ch;
      }
   }
   return buffer.toString( );
}
```

Draw control flow graph



Control flow graphs

- Be careful with:
  - Multi-way branches: if C or Java **switch** statements are used, the presence or absence of **break** statements alters the control flow
  - For loops: There are three separate parts: initialization, the loop test, and the "increment" statement(s).
  - Compound conditions: Is there short-circuit evaluation? If so, there is an implicit branch after each step of the evaluation.

Control flow graphs

- You may want to vary the level of precision depending on the situation.
- Does short circuit evaluation matter?

```
if ( a != null && a.someMethod() )
```

  yes – it prevents a null pointer exception

```
if ( a > 36 && a < 72 )
```

  order of evaluation is not likely to be relevant

- Should exceptions be considered?
  - In Java, exceptions other than those inheriting from `RuntimeException` must be declared – so you know where they can be thrown.
  - Exceptions inheriting from `RuntimeException` need not be declared:
    - Null pointer exceptions: possible for any `.` operator
    - Arithmetic exceptions: possible for arithmetic operators.

Purpose of the graph

- From a control flow graph, we can obtain potential execution paths
  - Some of the paths from the previous graph:

```
1 – 2 – 7
1 – 2 – 3 – 4 – 6 – 2 – 7
1 – 2 – 3 – 5 – 6 – 2 – 7
1 – 2 – 3 – 4 – 5 – 6 – 2 – 7
1 – 2 – 3 – 4 – 6 – 2 – 3 – 4 – 6 – 2 – 7
```

- We may decide to derive test cases to execute different paths through the program.
  - If there are an infinite number of paths, we will have to have some selection criteria to choose a subset of paths.
- For any selected path, we need to determine what data is needed to cause that path to be executed.
  - What value of `arg` will force the control flow to a selected path?

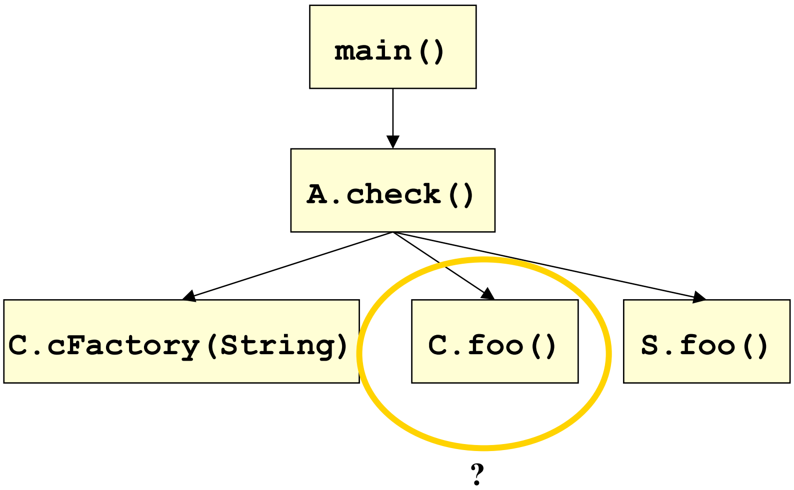
Interprocedural Call graphs

- Shows the relationships between calling and called methods.
- When you have polymorphism in an object-oriented language, determining the exact method called is no longer straightforward.
  - Is there a subclass that overrides a method?
  - Does the overridden method call the superclass method?
  - Are you using an object factory that returns objects of different types?

Example

```
public class C
{
   public static C cFactory( String kind )
   {
      if ( kind == "C") return new C();
      if ( kind == "S") return new S();
      return null;
   }

   public static void main( String[] args ) {( new A()).check() }

   void foo() { System.out.println("Parent foo() called"); }

   class S extends C
   {
      void foo() { System.out.println("Child foo() called"); }
   }

   class A
   {
      void check()
      {
         C myC = C.Factory("S" );
         myC.foo();
      }
   }
}
```

Potential call graph

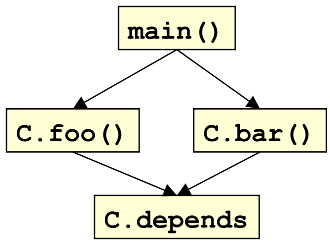


Call graphs

- With dynamic binding, we can only determine which of `C.foo()` and `S.foo()` will be called at run-time.
- We can read the code and, based on the parameters, determine that `S.foo()` is what would be called. However, a static code checker may not be able to make this determination.
- With static analysis, we have to consider the possibility that if a method `foo()` is called on a variable of type C, that the version of `foo()` executed may be in any
  - subclass of `C`, if `C` is a class
  - implementation of `C`, if `C` is an interface

Context and Call Graphs

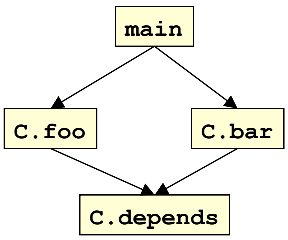
```
public class C
{
   public static void main( String[] args )
   {
      C myC = new C();
      myC.foo(3);
      myC.bar(17);
   }
   void foo( int n )
   {
      int[] myArray = new int[n];
      depends( myArray, 2 );
   }
   void bar( int n )
   {
      int[] myArray = new int[n];
      depends( myArray, 2 );
   }
   void depends( int[] a, int n )
   {
      a[n] = 42; // Can we tell if a[n] exists?
   }
}
```



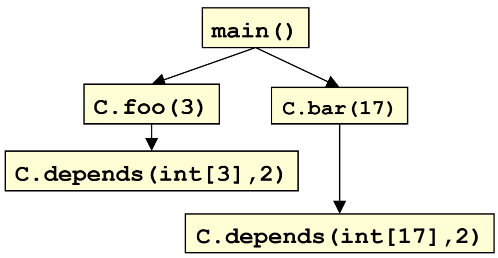
Context and Call Graphs

- A context-sensitive graph shows the parameter context in which a method is called.
  - In this case, the context-sensitive graph may help answer the question, "does **a[n]** exist?" in **depends()**.

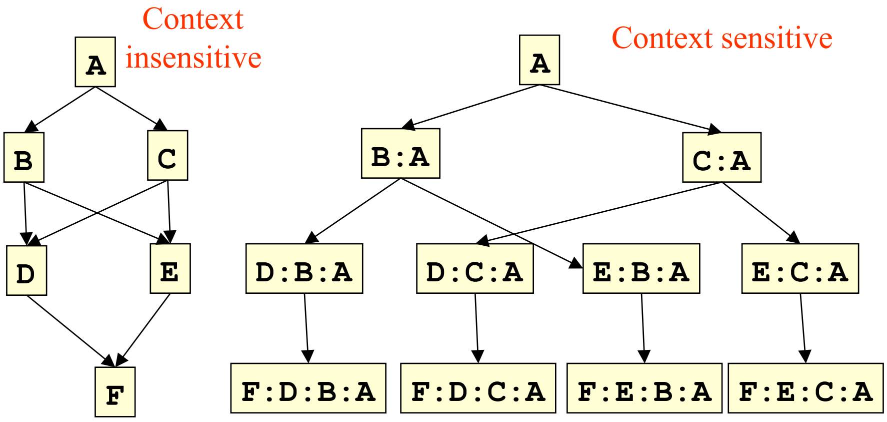
Context insensitive



Context sensitive



Paths in call graphs



- There are 4 distinct contexts in which method F may be called.
- The number of contexts can grow rapidly – even without recursion...

State Machines

Finite state machines

- In a finite state machine (FSM), there are:
  - a finite set of states S (nodes in a graph)
  - a finite input alphabet I, usually representing events
  - a finite output alphabet O, usually representing actions
  - an initial state $s_0 \in S$
  - a transition function: (graph arcs)
    - maps (a start state $\in S$ and an input $\in I$) to (a set of outputs $\in O$ and an end state $\in S$).
      - the start and end states may be the same
      - the set of outputs may be null, often indicated by a dash –.

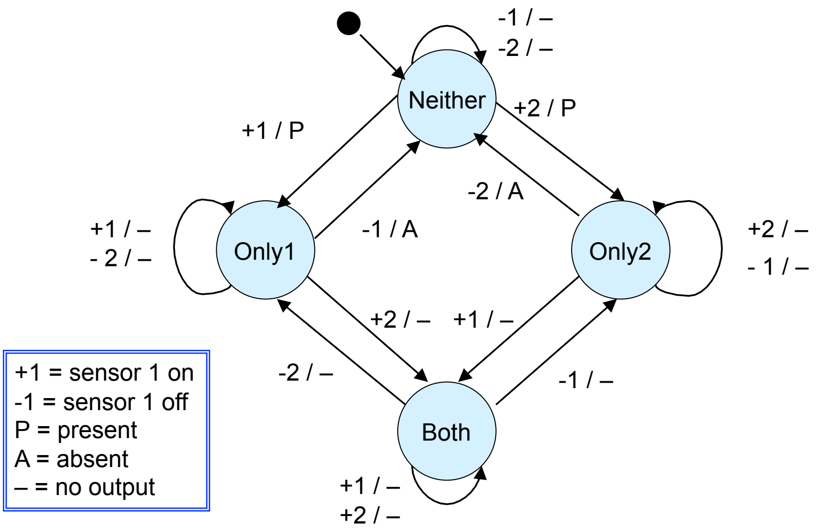
Finite state machines

- Strictly speaking, a FSM limited to inputs and outputs is a "Mealy machine".
  - There are no variables in such a model.
  - This model corresponds to a class of formal languages in theoretical computer science, and are related to regular expressions.
- Completeness property
  - For every state, there is a transition specified for every member of the input alphabet.
- Deterministic property
  - For every state, and for every member of the input alphabet, there is no more than one transition specified.
  - That is, for all known events in any state, it is predictable what the output and next states will be

FSM example: Door alarm sensors

- Two motion sensors for a room, one at door 1 and the other at door 2. When someone enters the detection range, a sensor reports (+). When someone leaves the detection range, a sensor reports (-).
- Integrate the two sensor inputs to produce:
  - P (present) event when someone enters the area
  - A (absent) event when the area becomes clear.

Sensor example



Sensor FSM in tabular format

Event State	+1	-1	+2	-2
None	Output: P NS: Only 1	NS: None	Output: P NS: Only 2	NS: None
Only 1	NS: Only 1	Output: A NS: None	NS: Both	Output: A NS: Only 1
Only 2	NS: Both	NS: Only 2	NS: Only 2	NS: None
Both	NS: Both	NS: Only 2	Output: P NS: Both	NS: Only 1

- NS = next state
- Advantage of this format: empty table cell means FSM is incomplete.

Uses for finite state machines

- Modelling event-driven systems
- Modelling systems where the sequence of actions is a major element of the system
- Lexical analysis
- Correctness checks applicable to FSMs:
  - Internal consistency: complete and deterministic
  - Paths through FSM satisfy some desired property, as specified from requirements or design.
  - A software implementation should conform to the FSM model.

Extended Finite State Machines

- A more generally applicable model is extended finite state machines (EFSMs).
- In an EFSM, variables are added to the FSM model.
  - Variables are associated with the entire EFSM.
  - Input events may have parameter values
  - Output events may have parameter values
  - Transitions can now do computations using the variables
  - Taking a transition may require that specified conditions hold on variables.

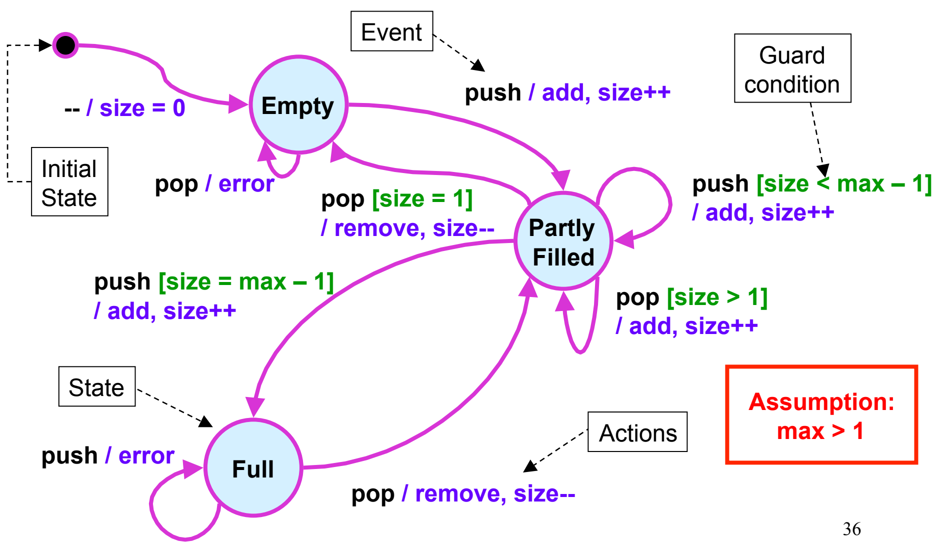
EFSM models

- An EFSM model consists of:
  - States: points where the system is waiting for an event to occur.
  - Variables: values associated with the state machine.
  - Transitions: the change from one state to another. A transition is composed of:
    - Event: stimulus to the system, that causes the implementation to exit from a state.
    - [optional] Guard condition(s): additional Boolean conditions that must be true to take the transition.
    - [optional] Actions: action(s) to perform as a result of the event
    - Next state: the new state that should be entered at the end of the transition.

State-based Models (2)

- An initial state must be identified.
  - A initial transition from the initial state to another state can be specified. This transition should not have an event or a guard condition; only action(s).
  - The implementation is in the initial state at start up, and immediately executes the initial transition without stimulus.
    - Analogy: calling a constructor method.
  - The implementation will never return to the initial state unless it is restarted; no transition can have the initial state as its next state.
- Actions can consist of:
  - Output events (this can trigger events in the same or other models)
  - Modification of state variables (if variables are used)

Example: A stack



Data Flow

Data flow models

- In a control flow model, we have been trying to capture the sequence of program actions.
- In a data flow model, we will try to model data dependencies.
- The basic data dependency is “definition-use” associations.
- Questions to answer:
  - Is there a use of a value before it has been defined?
  - Is a value defined, but never used?
  - At any use, do we have the value from the correct / expected definition?
  - Is optimizing the code possible, given the data dependencies?

Definition

- A definition is the point at which a variable receives a value for the first time, or has its previous value replaced with a new one.
- Where do definitions occur?
  - Left side of assignment statements
    - This includes shorthand statements such as `a++` which represents `a = a + 1`
  - Parameter assignment in method headers
  - "Input" statements
  - In some languages, at the point of declaration
    - Default values

Uses

- A variable is used whenever its value is obtained for some purpose.
  - Value is not changed
- Where can uses occur?
  - Expression evaluation
    - on the right side of an assignment statement
    - in a conditional statement
    - determination of an array index
    - return statements
  - Parameter passing at the calling side
  - "Output" statements
- Uses within conditional statements have a direct impact on the flow of control.

Definition-use associations

- A definition-use pair associates with each use, the definition that resulted in the current variable value.
  - There must be a potential execution path from the point of the definition to the point of the use
  - It is expected that the variable's value will not change during the execution of the path.
    - No other definitions of the variable along the path.

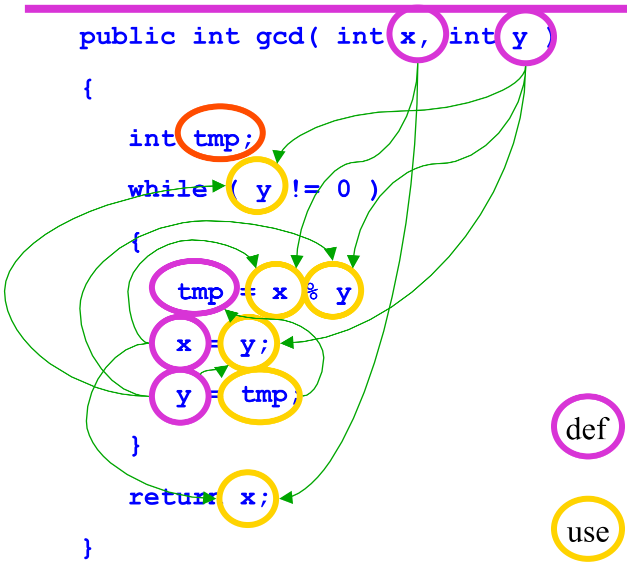
Example, again

```
public int gcd( int x, int y )
{
    int tmp;
    while ( y != 0 )
    {
        tmp = x % y;
        x = y;
        y = tmp;
    }
    return x;
}
```

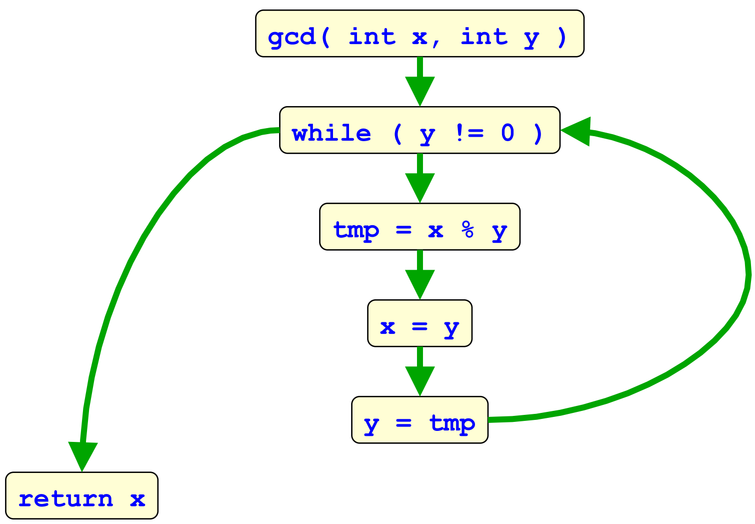
def

use

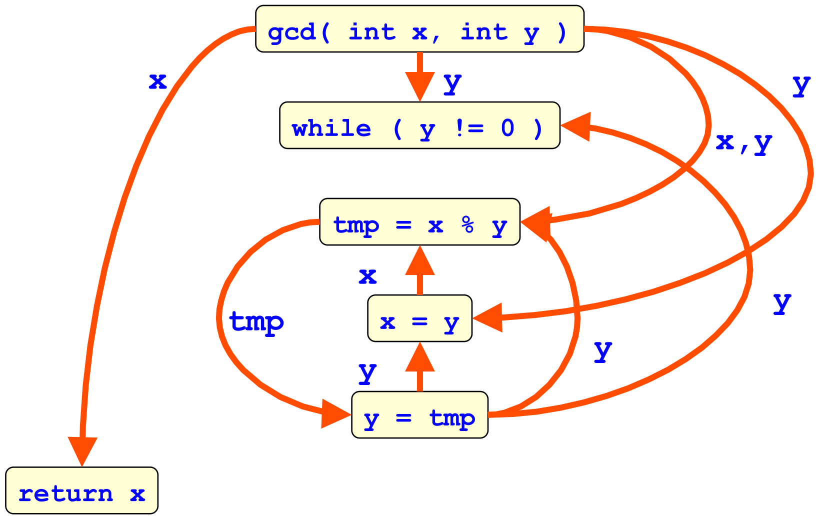
Example, again



Control flow graph



Data dependency graph



Appendix
Principles of Testing and Analysis

From S. Somé, A. Williams

Principles of Testing and Analysis

- Sensitivity
- Redundancy
- Restriction
- Partition
- Visibility
- Feedback

Sensitivity

- Sensitivity:
  - If a system could fail, how likely is it to actually do so?
  - Does the failure attract attention?
- A system that fails on a consistent and observable basis is more likely to have defects detected and removed.
- That is, "it is better to fail every time, than sometimes"

Example

- Three string copy routines:
- Copy string A to B, where memory for B has to be pre-allocated. The string A is too long to fit in B.
  - Version 1: write the string into memory without checking its length (like `strcpy(a,b)` in C). Memory beyond the boundary of B may be over-written.

```
while (((*b++)=(*a++))!='\0');
```

  - Version 2: if B has N characters, copy up to N characters into B (like `strncpy(a,b,n)` in C)

```
for (i=0;i<n,((*b++)=(*a++))!='\0';i++);
```

  - Version 3: check the length of A, and throw an exception if it is longer than B.

Sensitivity of the examples

- Version 1:
  - If the next variable in memory (let's say it is X) writes before B is read, then B may be read with an incorrect value.
  - If X is read, then it may have an incorrect value.
  - If subsequent memory is not used, there is no observable failure.
- Version 2:
  - If a string that is too long is written, it will be truncated. This may or may not be noticed by the application.
- Version 3:
  - If a string that is too long is written, an exception is thrown, which calls attention to the fault.

Redundancy

- We want to identify faults leading to differences between intended behaviour and actual behaviour.
- If one part of a system constrains the content of another, then the parts can be checked for consistency.
- Explicit declarations of intent: can be checked later.

Examples

- Type checking: a declaration of a variable to be of a certain type restricts the set of values that can be associated with the variable.
  - Later on, when an actual value is assigned, it can be checked against the declaration for consistency.
- Declaration of exceptions that might be thrown.
  - Ensures that calling methods are aware of the possibility.
- Checking consistency of implementation with specifications.

Restriction

- When the range of a property is too broad for effective checking, reduce the range of the property or check a reduced range subset.
- Architectural example:
  - Use of a stateless protocol means that a potential class of errors – being in the wrong state – is eliminated.

Example

- Use of integer code numbers for a set of choices.

```
static final int RED = 1;
static final int YELLOW = 2;
static final int GREEN = 3;
int trafficLightColour;
```

  - What happens if `trafficLightColour` has the value 4 (a legal `int` value)?
- Use of restriction:

```
enum Colour { RED, YELLOW, GREEN; }
Colour trafficLightColour;
```

  - This is a clearer declaration of intent of the purpose of the variable.
  - With this approach, the type is now restricted to the three legal values, and the compiler can check for validity.

Example

```
static void questionable()
{
   int k;
   for ( int i = -; i < 10; i++ )
   {
      if ( someCondition(i) )
      {
         k = 0;
      }
      else
      {
         k = k + i;
      }
   }
   System.out.println( k );
}
```

- Static analysis: determine properties from code without running the code
- Could a static analysis tool answer the question "is **k** initialized?"
- What about the restricted property: "is there a possibility that **k** might not be initialized?"

Partition

- "Divide and conquer"
- Partitioning can be a useful principles from several viewpoints:
  - Partition a complex system into several sub-systems, and test the parts. Then, integrate the parts.
  - Partition a range of inputs into groups of values which should exhibit similar behaviour:
    - Example: for the absolute value function for an integer input, partition the domain into negative integers, zero, and positive integers.
      - Now there are 3 cases instead of the 4+ billion possible integers.

Visibility

- The ability to gain access to information.
  - Also, the ease of access to the information.
- While it is a good design principle to employ information hiding, the same principle can make a system more difficult to test.
  - Anything that is exposed to view can be checked during test execution.
  - Anything that is not accessible cannot be checked directly.
    - It may be possible to collect such information indirectly.

Examples of visibility

- Choice of a text-based protocol versus a bit-based protocol.
  - If the performance tradeoff is acceptable, use of a human-readable protocol can pay off in terms of being able to easily construct test messages and check their correctness.
- Exposing an interface
  - This allows a potential test access and observation point.
- Consider testing a public method versus a private method.
  - A public method can be called directly.
  - There is no direct access to a private method. If you want to test such a method, how do you arrange for it to be called?

Feedback

- Using data that is collected from a process to modify the process.
- Examples for testing:
  - Choosing tests to run based on results of previous tests.
    - "If test 47 passes, go to test 48. If test 47 fails, run tests 47A and 47B."
  - Using historical data to target test effort:
    - "Class A tends to have more defects than other classes. Add additional test cases for this class."
  - Post-mortem analysis: After a software version is released, use customer problem reports to identify areas of process improvement.

Summary of principles

- Sensitivity: better to fail every time than some times.
- Redundancy: make intentions explicit.
- Restriction: make the problem easier.
- Partition: divide and conquer
- Visibility: make information accessible
- Feedback: apply lessons from experience.