DIT635 - Mutation Testing Exercise

First, if you have not finished the activity from Lecture 12 (Mutation Testing), do so!

In a previous exercise, you wrote unit tests for a Meeting Planner system based on both the functionality and code structure. We will not return to the Meeting Planner one last time to assess the sensitivity of your test cases to seeded faults in the code.

1. Generate at least four mutants for classes of your choice in the MeetingPlanner code.
    a. You must create at least one invalid, one valid-but-not-useful (non-equivalent), one useful, and one equivalent mutant.
    b. Each mutant must be created by applying a different mutation operator, and you must use at least one mutation operator from each of the three categories in the attached handout.
    c. You do not have to use the same classes or methods for all mutant categories. Try mutating different parts of the code.
2. Assess your test suite that you created in previous exercises, with respect to the set of mutants that you derived - Are you able to kill all of the non-equivalent mutants with your test suite? If not, write additional tests that can kill those non-equivalent mutants.
3. Identify a minimal subset of tests from your test suite that is sufficient to kill all of the non-equivalent mutants.

If you finish early, try adding mutations to the CoffeeMaker classes from Homework Assignment 1. Do your unit tests detect those mutations?

ID	Operator	Description	Constraint
Operand Modifications			
crp	constant for constant replacement	replace constant $C1$ with constant $C2$	$C1 \neq C2$
scr	scalar for constant replacement	replace constant C with scalar variable X	$C \neq X$
acr	array for constant replacement	replace constant C with array reference $A[I]$	$C \neq A[I]$
scr	struct for constant replacement	replace constant C with struct field S	$C \neq S$
svr	scalar variable replacement	replace scalar variable X with a scalar variable Y	$X \neq Y$
csr	constant for scalar variable replacement	replace scalar variable X with a constant C	$X \neq C$
asr	array for scalar variable replacement	replace scalar variable X with an array reference $A[I]$	$X \neq A[I]$
ssr	struct for scalar replacement	replace scalar variable X with struct field S	$X \neq S$
vie	scalar variable initialization elimination	remove initialization of a scalar variable	
car	constant for array replacement	replace array reference $A[I]$ with constant C	$A[I] \neq C$
sar	scalar for array replacement	replace array reference $A[I]$ with scalar variable X	$A[I] \neq X$
cnr	comparable array replacement	replace array reference with a comparable array reference	
sar	struct for array reference replacement	replace array reference $A[I]$ with a struct field S	$A[I] \neq S$
Expression Modifications			
abs	absolute value insertion	replace e by $\mathsf{abs}(e)$	$e < 0$
aor	arithmetic operator replacement	replace arithmetic operator ψ with arithmetic operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
lcr	logical connector replacement	replace logical connector ψ with logical connector ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
ror	relational operator replacement	replace relational operator ψ with relational operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
uoi	unary operator insertion	insert unary operator	
cpr	constant for predicate replacement	replace predicate with a constant value	
Statement Modifications			
sdl	statement deletion	delete a statement	
sca	switch case replacement	replace the label of one case with another	
ses	end block shift	move $\}$ one statement earlier and later	

Figure 16.2: A sample set of mutation operators for the C language, with associated constraints to select test cases that distinguish generated mutants from the original program.