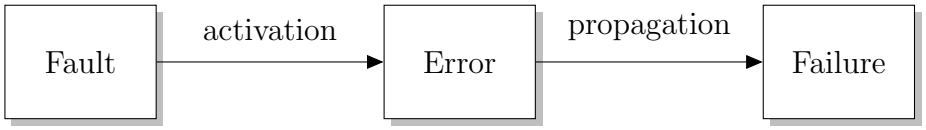
Task 1. - Fault Tolerance

a) A **failure** has occured when the external behaviour of a system deviates from that which is specified for it. Failures result from unexpected problems in the system internals that manifest themselves in the system's external behaviour, these problems are called **errors**. The underlying mechanical or algorithmic cause of an error is called a **fault** in the program.



b) It is impossible to determine every single way a system might fail, and develop error handling procedures for each different failure mode. Instead, consider what the worst case error that needs to be handled is, and then treat every other error as this type of error. This technique is refered to as **merging of error modes**, and simplifies error handling, but might result in some errors having more drastic consequences than strictly necessary. Merging of error modes also simplifies error detection, since knowing the cause of the error is not needed to handle it.

c) A system is **failfast** if it stops execution when it detects a fault, and additionally if the time between the occurrence of a fault (the error) and its detection (the failure) is small. A failfast system will immediately stop (crash) and restart when a failure is detected, and the philosphy is that it is better to do nothing than continue the computation in some unspecified way, which means that the consequences of the error doesn't spread to other modules.

d) **Fault prevention** attempts to eliminate any possibility of faults in a system before it goes operational, and compasses techniques such as unit/software inspections, verifications and testing. The problem with this approach is that you can only test for the presence of bugs, not the absence. It is impossible to test for everything that can go wrong, additionally hardware components will still fail over time no matter how much testing of the system has been performed. The alternative is **fault tolerance** which enables a system to continue functioning even in the presence of faults. Instead of testing for specific faults, the system uses acceptance tests to check if anything is wrong during execution.

If an issue is detected, the system handles the error using techniques such as merging of error modes, which enables the system to function reliable even in the presence of errors.

e) The different tests that can be implemented are:

- Replication Checks
- Timing Checks
- Reversal Checks
- Coding Checks
- Reasonableness Checks
- Structural Checks
- Dynamic Reasonableness Checks

See Burns & Wellings Chapter 2, page 42, for more details.

f) The two main strategies for error recovery are

- **Backward error recovery**: Stores recovery points after successful acceptance tests containing the full (and consistent) state of the system. If an acceptance test fails at a later point, the system is restored to the previous safe state (from the recovery point) and executes an alternative section of the program.
- **Forward error recovery**: When an acceptance test fails, the system attempts to continue from an erroneous state by making selective corrections to the system state to arrive at a consistent state. Simultaneously the system makes safe any aspect of the controlled environment which may be hazardous or damaged because of the failure.

g) **Asynchronous transfer of control** is used in combination with forward error recovery to notify other participants in an action that they need to do error recovery. The thread detecting the error will interrupt the other threads in what they do, and error handling starts immediately.

- **C/POSIX**: There is no built-in support for this in the language, but you can still dynamically create and cancel threads. Specifically, threads can be terminated asynchronously, but the thread itself controls the access to this. Asynchronous transfer of control can be implemented using `setjmp/longjmp`, along with termination.

- **Java**: Standard Java supports

```
public void interrupt() throws SecurityException
public Boolean isInterrupted()
```

  This requires that the interruptible thread polls with `isInterrupted()`, which is not good for Real-Time applications. Real-Time Java allows for exceptions being thrown to the interrupted thread instead of polling using `AsynchronouslyInterruptedException`. However, exceptions are difficult to deal with in its basic form, and gets way worse with multithreading. Like `setjmp/longjmp`, this is a very inelegant solution due to its implementation.

- **Ada**: Ada makes it easy with its `select then abort` statement such as

```
select
  delay 1.0 -- or any event.
then abort
  -- Do the real work
  -- Will be interrupted if it times out or the event
    happens.
end select;
```

  The normal execution of code happens inside the `abort` block, this code execution will be aborted if the `select` statement receives an event.

h) The domino effect happens when a threads need to revert to a recovery point after detecting an error, but having exchanged possibly inconsistent data during inter-process communication with another thread after the recovery point. Consider figure 1, if an error occurs at time T_e in thread P_2, the thread needs to revert to recovery point R_{22}. However, the IPC_4 also needs to be undone since the data might be inconsistent, so P_1 needs to revert to R_{12} as well. This leads to IPC_3 needing to be reverted, and so on. As is the case with the figure, the worst case of the domino effect is that both threads need to roll back their state all the way to the start of the computation, which might be infeasible for a real-time system.

i) Transactions are a way to synchronize error handling between multiple participants involved in an action. A transaction is an action that

- Keeps track of participants in the action
- Defines a consistent "starting point" for the action
- Enforces borders for the action

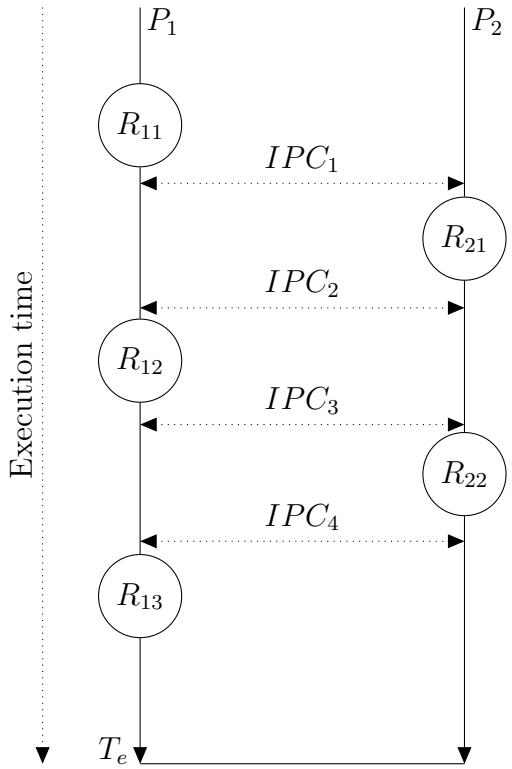


Figure 1: The domino effect

- Ensures that the action is consistent when completed

and has the following properties

- **Atomicity**: The transaction completes successfully (commits) or if it fails (abort) all of its effects are undone (rolled back)
- **Consistency**: Transactions produce consistent results and preserve application-specific invariants
- **Isolation**: Intermediate states produced while a transaction is executing are not visible to others. Furthermore, transactions appear to execute serially, even if they are actually executed concurrently
- **Durability**: The effects of a committed transaction are never lost (except by a catastrophic failure)

Transactions prevent the domino effect by synchronizing the recovery points of all participants at the start of the action, and at the end of a successfully committed action. This ensures that a backward error recovery only needs to roll back to the start of the transaction to be consistent.

j) The **Two-Phase Commit** protocol is the mechanisms the transaction manager uses to ensure that all participants in the action agrees on the outcome. The manager asks each of the participants if they are ready to commit, and they each answer commit/fail depending on if they encountered error. If the transaction manager receives a fail (or nothing at all from a participant), the transaction has failed, and the manager orders a roll back. Otherwise, the manager informs all participants to commit the action. A sequence diagram and illustration of the protocol is shown in figure 2 and 3.

k) The three cases are

- **Storage**:
  - Static redundancy: Probability failure rates specifies the required redundancy to ensure availability according to specification. Use n copies of storage modules (hard drives). Read and write to all redundant modules, the probability for n bad reads so small that it can be ignored. To ensure consistent data, compare version id and use data with newest id

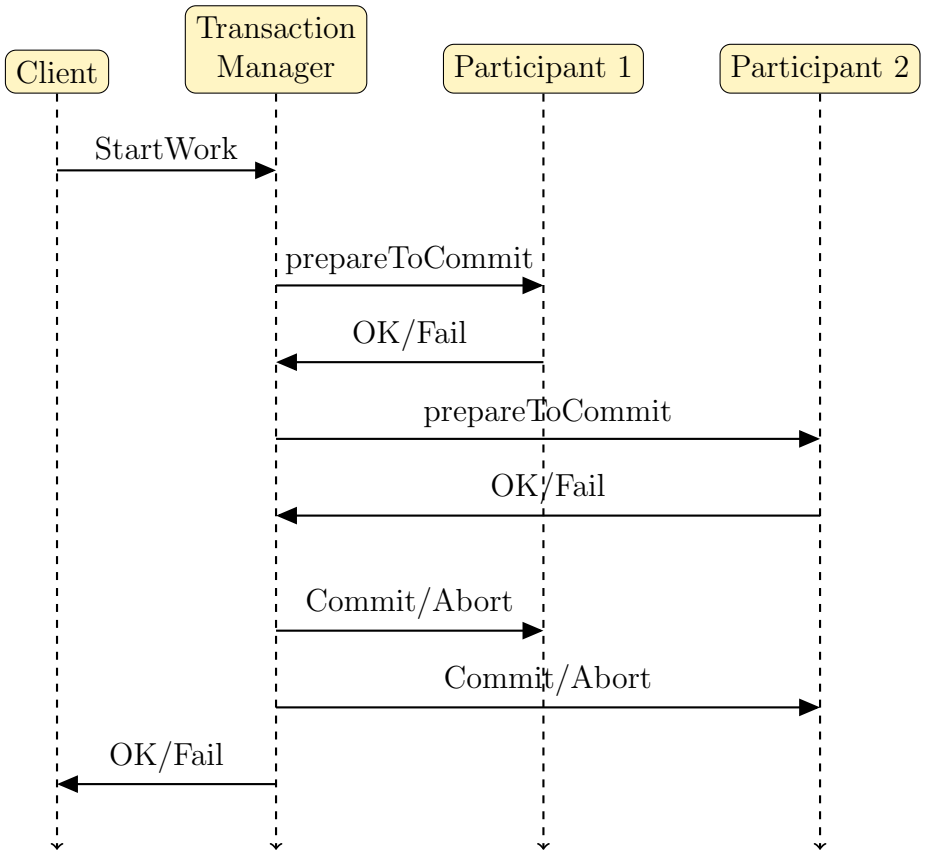


Figure 2: Sequence diagram for a Two-phase Commit

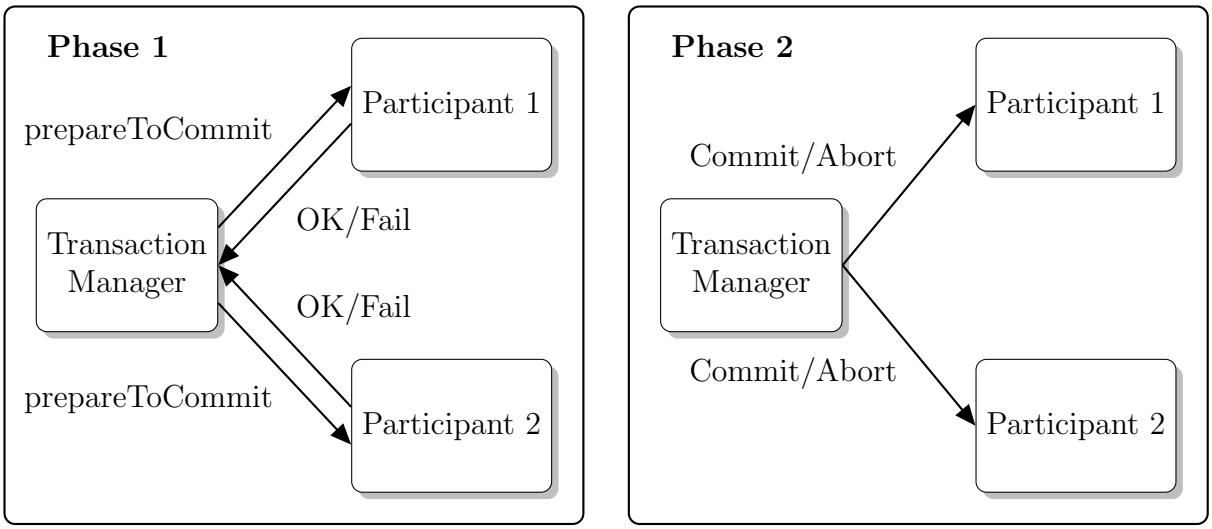


Figure 3: Illustration of a Two-Phase Commit protocol

  - Dynamic redundancy: All reads leads to write-back on error, assuming that at least one storage module returned valid data. Repair thread reads all pages regularly and performs write-backs as required. Data used seldom might deteriorate due to age, repair thread refreshes the pages

- **Communication**: Dynamic redundancy implemented using a timer and resend mechanisms. After sending message, start timer and wait for ack back from receiver, and then resend the message on absent ack. Threat all errors as message lost and resend.
- **Calculation**: Three different ways of handling with redundancy
  - Checkpoint - restart: Similar to backwards recovery using recovery points. Calculation is executed in steps
    1. Calculate result and side effect
    2. Acceptance test, if it fails, restart from previous stored checkpoint of last execution
    3. Store full state as recovery point in reliable storage, and write the complete state to reliable storage module
    4. Do side effect, in case of restart, previous side effect might be executed twice
  - Process Pairs: Two processes, Primary & Backup. Primary performs the work and sends periodic "I'm alive" messages to Backup, and sends full state to Backup after state change. Backup is the checkpoint, takes over when the Primary fails. Failfast implementation, Primary instant crash and restart on failure. Masks hardware failure (processor failures) as well as transient software failures (Heisenbugs). Redundancy by resending masks communication errors.
  - Persistent processes: Assumes transactional infrastructure with databases to preserve data and operating system that implement transactions. All calculations are transactions, atomic transformations from one consistent state to another, either done completely or not at all. The processes are stateless with all data safely stored in highly available databases. First read data from database, then before side effects, write updated data back to database

Task 2. - Shared Variable Synchronization

a) The standard problems that are associated with sharing memory between threads are

- Deadlock: The system is locked in a circular wait. Example: waiting for mutual resources, or waiting for lost message.
- Livelock: The system is locked in a subset of states. More general form of deadlock, but in a sense the opposite. Harder to diagnose than deadlock since the program is doing work. Example: sending messages requiring acks,
- Starvation: A thread does "by accident" not get necessary resources. A resource could be a semaphore, or more abstract such as CPU time. Leads to the thread getting to run less than it needs to. Could be caused by bad luck, unfair scheduling, or timing cycles where the same thread ends up last in the priority queue every time. Example: multiple threads requiring dynamic memory from the heap. One thread uses a lot of memory over a short time, multiple other threads require small amounts of memory all the time. Might never be enough available memory for the first thread to run
- Race conditions: A bug that surface by unfortunate timing or order of events

b) The rendezvous can be implemented using two semaphores both initialized to 0:

```
1  rendezvousWait1 = Semaphore(0)
2  rendezvousWait2 = Semaphore(0)
3
4  void t1_run(){
5    doWork(t1);
6
7    rendezvousWait2.signal()
8    rendezvousWait1.wait()
9
10   exit(t1);
11 }
12
13 void t2_run(){
14   doWork(t2);
15
16   rendezvousWait1.signal()
17   rendezvousWait2.wait()
18
19   exit(t2);
20 }
```

The order of the statement are fairly irrelevant as long as not both threads wait before signalling.

c) A race condition is a bug that surfaces by unfortunate timing or order of events. The following are race conditions in the stocker/stacker problem:

- The mutual exclusion implementation at line 7-8 and line 22-23 fails when the thread gets interrupted right after checking the `woodPileBusy` flag, but before setting `woodPileBusy=true`. Both threads could then run its critical region simultaneously causing data inconsistencies.
- The stoker thread will suspend itself after testing for an empty wood pile in line 9, but might be interrupted right before the call to `suspend()` in line 14. The stacker then might run its code which refills the woodpile and then resumes the stoker thread, before the stoker thread is suspended. This causes the stoker thread to enter a permanent suspended state, while the stacker is stuck in a livelock where it never has an empty woodpile to refill.

There might be even more errors, it's a badly written piece of code.

d) Suggested solution using semaphores:

```
1  bool woodPileEmpty = false;
2  int amountWoodInPile = 10;
3
4  woodPileMutex = semaphore(1);
5  woodPileRefilled = semaphore(0);
6
7  void stoker(){
8    while(1){
9      wait(woodPileMutex);
10
11     if(amountWoodInPile > 0){
12       amountWoodInPile -= 1; // Adding wood to fire
13       signal(woodPileMutex);
14     } else {
15       woodPileEmpty = true;
16       signal(woodPileMutex);
17       wait(woodPileRefilled);
18     }
19   }
20 }
21
22 void stacker(){
```

```
23    while(1){
24      wait(woodPileMutex);
25      if(woodPileEmpty){
26        amountWoodInPile = 10; // Refilling woodpile
27        woodPileEmpty = false;
28        signal(woodPileMutex);
29        signal(woodPileRefilled);
30      } else {
31        signal(woodPileMutex);
32      }
33    }
34  }
```

e) Pseudocode from *Little Book of Semaphores* page 157, using the semaphores and variables given by the hint in the task description:

```
1  void car_thread(){
2    load()
3    boardQueue.signal(C)
4    allAboard.wait()
5
6    run()
7
8    unload()
9    unboardQueue.signal(C)
10   allAshore.wait()
11 }
12
13 void passenger_thread(){
14   boardQueue.wait()
15   board()
16
17   mutex.wait()
18     boarders += 1
19     if boarders == C:
20       allAboard.signal()
21       boarders = 0
22   mutex.signal ()
23
24   unboardQueue.wait()
25   unboard()
26
27   mutex2.wait()
28     unboarders += 1
29     if unboarders == C:
30       allAshore.signal()
31       unboarders = 0
32   mutex2.signal()
33 }
```

f) Suggested solution using Java, with a focus on preserving the spirit of the problem from the semaphore implementation. That is, the first 4 threads arriving get to board the roller coaster car together, and then unboard together. A simpler solution with just two barriers is also acceptable.

```
class Rollercoaster
{
  private int waitingToBoard;
  private bool boarding;

  Rollercoaster()
  {
    waitingToBoard = 0;
    boarding = false;
  }

  synchronized rideRollercoaster(){
    while(waitingToBoard > 4 || boarding) wait(); // Queue
    for the coaster if 4 are ready to board

    waitingToBoard++;
    if(waitingToBoard == 4) boarding = true;

    while(!boarding) wait(); // Wait here for boarding until
    4 are ready

    waitingToBoard--;

    // Rollercoaster work for each thread happens here
    notifyAll();

    while(waitingToBoard > 0) wait(); // Wait here until
    rollercoaster work is completed for all threads

    if(waitingToBoard == 0) boarding = false
    notifyAll();
    }
  }
}
```

g) Suggested solution using Ada, with a focus on preserving the spirit of the problem from the semaphore implementation. That is, the first 4 threads arriving get to board the roller coaster car together, and then unboard together. A simpler solution with just two barriers, and without requeue, is also acceptable.

```
protected type RollerCoaster is
```

```
2   entry enterQueue;
3 private
4   entry boardingQueue;
5   entry unboardingQueue;
6   boarding : Boolean : False;
7   unboarding : Boolean : False;
8   waitingToBoard : Integer : 0;
9   seatsInCar : constant Integer : 4;
10 end RollerCoaster;
11
12 protected body RollerCoaster is
13   entry enterQueue when not carFull is
14   begin
15     waitingToBoard := waitingToBoard + 1;
16     if waitingToBoard = seatsInCar then
17       carFull := True;
18     end if;
19     requeue boardingQueue
20   end enterQueue;
21
22   entry boardingQueue when boardingQueue'Count = seatsInCar
       or boarding is
23   begin
24     boarding := True;
25     waitingToBoard := waitingToBoard - 1;
26     if boardingQueue'Count = 0 then
27       boarding := False;
28     end if;
29
30     -- Rollercoaster work for each thread happens here
31
32     requeue unboardingQueue;
33   end boardingQueue;
34
35   entry unboardingQueue when unboardingQueue'Count =
       seatsInCar or unboarding is
36   begin
37     unboarding := True;
38     if unboardingQueue'Count = 0 then
39       unboarding := False;
40       carFull := False;
41     end if;
42   end unboardingQueue
43 end RollerCoaster
```

Task 3. - Scheduling

a)

- **Benefits**: There are no concurrent tasks existing at run-time, only a set of procedures that are executed sequentially, which makes the system easier to understand and maintain. The procedures all share a common address space, and can easily exchange information between themselves. This data does not need to be protected, since concurrent access is not possible, and the system therefore avoids the standard problems connected to shared variable synchronization.
- **Drawbacks**: All task periods must be a multiple of the cycle time (remember the problem with the blinking LEDs with different periods). It is difficult to incorporate activities with periods longer than the cycle time, and sporadic activities are impossible to incorporate. Tasks with a considerable computation time needs to be split into smaller procedures with fixed size, which are more susceptible to implementation errors.

b) There are $N = 3$ tasks in the task set with utilization given by

$$U \equiv \sum_{i=1}^{N} \frac{C_i}{T_i} = \frac{8}{16} + \frac{2}{8} + \frac{1}{4} = 1.0, \tag{1}$$

which is greater than the threshold value for 3 tasks

$$N\left(2^{\frac{1}{N}} - 1\right) = 3\left(2^{\frac{1}{3}} - 1\right) = 0.78, \tag{2}$$

and we can therefore not conclude that the system is scheduable from the test. Note that this only means that we can't conclude, since the test is sufficient, but not necessary.

c) The time-line for the task set is shown in figure 4. The color of the task intervals indicates if it's preempted (blue) by a task with higher priority or executing (green) currently. The arrows indicate release time for each of the tasks, the green circle indicates that the task met its deadline. Clearly the task set is able to meet all it's deadline and is schedulable. This does not conflict with the previous result from the utilization test, since the test is only sufficient, but not necessary.

d) **Priority inversion** can occur in systems where threads with different priorities all use the same resources. When a lower priority task holds the resource it effectively blocks higher priority tasks from acquiring the resource (and therefore execute), for as long at the resource is being held. The lower priority thread then effectively runs on a higher

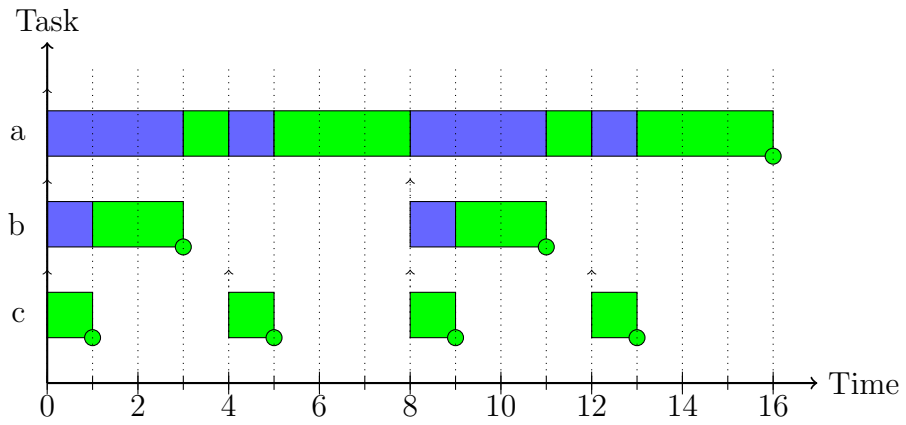


Figure 4: Time-line for the task set

priority then the high priority threads. There are two cases of priority inversion

- **Bounded priority inversion**: The lower priority task will at some point finish its work and release the resource, allowing the higher priority threads to execute under normal priority hierarchy. This often occurs in real-time systems and are usually not a big problem.
- **Unbounded priority inversion**: The higher priority thread never gets to run since the lower priority thread never gives up the shared resource, which might happen if it itself is blocked by a higher priority thread. This can cause deadlocks, or more specifically that the higher priority thread (which is most likely time-sensitive and critical) never gets to run.

e) The unbounded priority inversion problem can be solved using a **Priority Ceiling protocol.** The algorithm allocates a priority value to the shared resources in the system, according to the highest priority of all the tasks wishing to interact with the resource. A thread allocating the resource then inherits the resource's priority (or really: `max(current,resource)`) while it owns it. Alternatively, **Priority Inheritance** can be used where the blocking low-priority task is assigned (inherits) the priority of the blocked high-priority task.

Task 4. - Code Quality

a) Referring to the checklists from *Code Complete*, the following can be regarded as weaknesses in the interface of the module

- The modules doesn't have a central purpose, it control walking speed and also provides information about sensor data.
- The module name `WALKING.H` implies that the module handles all aspect of walking, not only walking speed. Additionally it doesn't take into account the sensor data output.
- The abstraction in the module is not consistent, providing both high level routines for setting speed (`SetWalkingSpeed`) as well as low-level routine using bytes (`SetLeftMotor`).
- It is not obvious how you use the module since it provides a number of routines that seem similar (`StartLocomotion`, `SetWalkingSpeed`, `SetLeftMotor`). It is also not clear how the `PrintStatus` procedure works since it requires the passing of a `TStatus` type variable.
- Unclear if `int & gyroStatus` is an input or output parameter to `getWalkingSpeed`? Either way, what does it have to do with walking speed? Furthermore `getWalkingSpeed` returns a `float` and not a `double`, this is inconsistent.
- The routine `GetStatus` uses a global semaphore, and the implementation of this routine needs to be taken into account whenever it's used in connection with other modules using the same semaphore
- The `TStatus` type is not encapsulated and is accessible through the interface. The members are also poorly named and seem unrelated to each other.
- The routines are not consistently named with their opposites (`StartLocomotion`, `Exit`). Why does `SetLeftMotor` and `SetRightMotor` use two different input types when they're named as a pair?
- The interface might not be minimal, both `getStatus` and `getWalkingSpeed` returns walking speed.

b) The following list is taken from *Code Complete* page 162:

- The routine has a bad name. `HandleStuff()` tells you nothing about what the routine does.
- The routine isn't documented.

- The routine has a bad layout. The physical organization of the code on the page gives few hints about its logical organization. Layout strategies are used haphazardly, with different styles in different parts of the routine. Compare the styles where `expenseType == 2` and `expenseType == 3`.
- The routine's input variable, `inputRec`, is changed. If it's an input variable, its value should not be modified (and in C++ it should be declared `const`). If the value of the variable is supposed to be modified, the variable should not be called `inputRec`.
- The routine reads and writes global variables — it reads from `corpExpense` and writes to profit. It should communicate with other routines more directly than by reading and writing global variables.
- The routine doesn't have a single purpose. It initializes some variables, writes to a database, does some calculations — none of which seem to be related to each other in any way. A routine should have a single, clearly defined purpose.
- The routine doesn't defend itself against bad data. If `crntQtr` equals 0, the expression `ytdRevenue * 4.0 / (double) crntQtr` causes a divide-by-zero error.
- The routine uses several magic numbers: 100, 4.0, 12, 2, and 3.
- Some of the routine's parameters are unused: `screenX` and `screenY` are not referenced within the routine.
- One of the routine's parameters is passed incorrectly: `prevColor` is labelled as a reference parameter (&) even though it isn't assigned a value within the routine.
- The routine has too many parameters. The upper limit for an understandable number of parameters is about 7; this routine has 11. The parameters are laid out in such an unreadable way that most people wouldn't try to examine them closely or even count them.
- The routine's parameters are poorly ordered and are not documented.

Additionally, the following can be seen as weaknesses

- The variable and parameter names in the function are abbreviated too much, for example `crntQtr` could be called `currentQuarter` for clarity. Additionally the abbreviation style is inconsistent.

- The iterator `i` has a confusing name, especially since it's used as something besides an iterator in line 17 and 20.
- The `status` variable doesn't seem to have a purpose since it's always set to `SUCCESS`, this could perhaps have been a simple bool?