Marcel ZAUDER 16-124-836
Pascal GERIG 16-104-721

4.1 Several Questions

a) *Why are immutable classes inherently safe?*
Immutable classes are only instantiated once and its state cannot be changed after that. Therefore they are inherently safe because threads that have access to this class only read its state but are unable to change it.

b) *What is "balking"?*
The balking pattern is a software design pattern that only executes an action on an object when the object is in a particular state. It is used when objects are generally only in a state that is prone to balking temporarily but for an unknown amount of time. The object will make itself not available and will return any call instantly (in Java there will be an exception thrown) while being in an "incorrect" state until it reaches a correct state again, i.e. A object wants to read a text file and upon a calling method invoked by another object wants to access information via a get method an that object, the object would "balk" at this request.

c) *When is partial synchronization better than full synchronization?*
In partial synchronization only the "critical sections" are synchronized. This is useful when objects have both mutable and immutable instance variables or when methods can be split up in parts that are not critical sections and ones that are.

d) *How does containment avoid the need for synchronization?*
Containment indicates that one object contains another. Because of this these contained objects are conceptualized as exclusively held resources that are not needed for shared access. Because these objects are "unshared" there is no need for synchronization because the contained objects are always held by one object exclusively.

e) *What liveness problems can full synchronization introduce?*
The most often occuring liveness problem is the one of starving because in a fully synchronized object every method is synchronized. Therefore it can happen that if one thread wants to access this particular object it could be always "busy" and therefore the thread will wait "for eternity" until no other thread occupies that object.

f) *When is it legitimate to declare only some methods as synchronized?*
For example when one method is only called by another method only the calling method needs to be synchronized because the other method is only accessed from that one and therefore it will have a synchronized behaviour. Also a method that does not change the state of an object does not necessarily need to be synchronized if it accesses a state of an object that is immutable.

4.2 Dining Savages

```
const M = 5

SAVAGE = (getserving -> SAVAGE).
COOK = (fillpot -> COOK).

MEAL = LARGEPOT[M],
LARGEPOT[n:0..M] = (when (n>0) getserving -> LARGEPOT[n-1]
                        | when (n==0) fillpot -> LARGEPOT[M]).

||DININGSAVAGES = (SAVAGE || COOK || MEAL).
```

Marcel ZAUDER 16-124-836
Pascal GERIG 16-104-721

4.3 Lift

```
property LIFTCAPACITY = LIFT[0],
LIFT[i:0..8] = (enter -> LIFT[i+1]
                |when(i>0) exit -> LIFT[i-1]
                |when(i==0)exit -> LIFT[0]
                ).
```

a) *Which values can the variable i have in non-error states?*
The variable *i* can be all the values from 0 to 8 so the FSP is in a non-error state.

b) *What kind of property is used and what does it guarantee?*
This property is a liveness property which guarantees that the LIFT[0] state can always be reached.

c) *Provide an action trace that violates the provided property.*
An action trace that violates the provided property would be:
enter -> enter -> enter -> enter -> enter -> enter -> enter -> enter -> enter
Here we would enter 9 times and therefore *i* would be 9 which would be not in the scope of the variable definition.

d) *Provide an action trace that does not violate the provided property.*
An action trace that does not violate the provided property would be:
enter -> exit -> enter -> enter -> exit -> exit -> enter -> exit -> enter

4.4 Thread-Safe MessageQueue

See the attached MessageQueue files. We are using the Containment pattern, because *full-synchronization* is not a particular fitting solution because of the edge cases, that we cannot add to a full queue and remove from an emtpy queue, which would need balking, which would lead to a very "ugly" implementation. Therefore we decided to use the containment pattern, because for *partial synchronization*, which would be possible to implement, there is too little code to justify using it.