Mark Gameng

CS 487 – Spring 2022

Homework 3

1. Reusable exception management
    a. **Assess the appropriateness of a value read from a sensor**

```
Def temp_check(thermometer):

        // checks temperature and assess appropriateness. If not appropriate remove equipments and thermometer

        temp = int(thermometer.value)

        if temp > 100: out = (temp, “Really hot”)

        elif temp > 70: out = (temp, “hot”)

        elif temp > 50: out = (temp, “normal”)

        elif temp > 30: out = (temp, “cold”)

        else: out = (temp, “really cold”)


        if out[1] in [“really hot”, “really cold”]: // only want temp values not really cold or hot

                raise Exception // remove thermometer and other equipments

        return out // appropriate
```

    b. **Reusable module which would make an inappropriate value appropriate**

Imagine the method in part a being used to remove the thermometer and equipment if the place is too hot or cold. For example, if its too hot or cold, the thermometer might stop working or break and cause heavy damage to other equipment. However, if the thermometer or equipment becomes more advanced and able to take in more extreme conditions, then the method in part a would need to be change. The inappropriate value from part a, can then be appropriate.

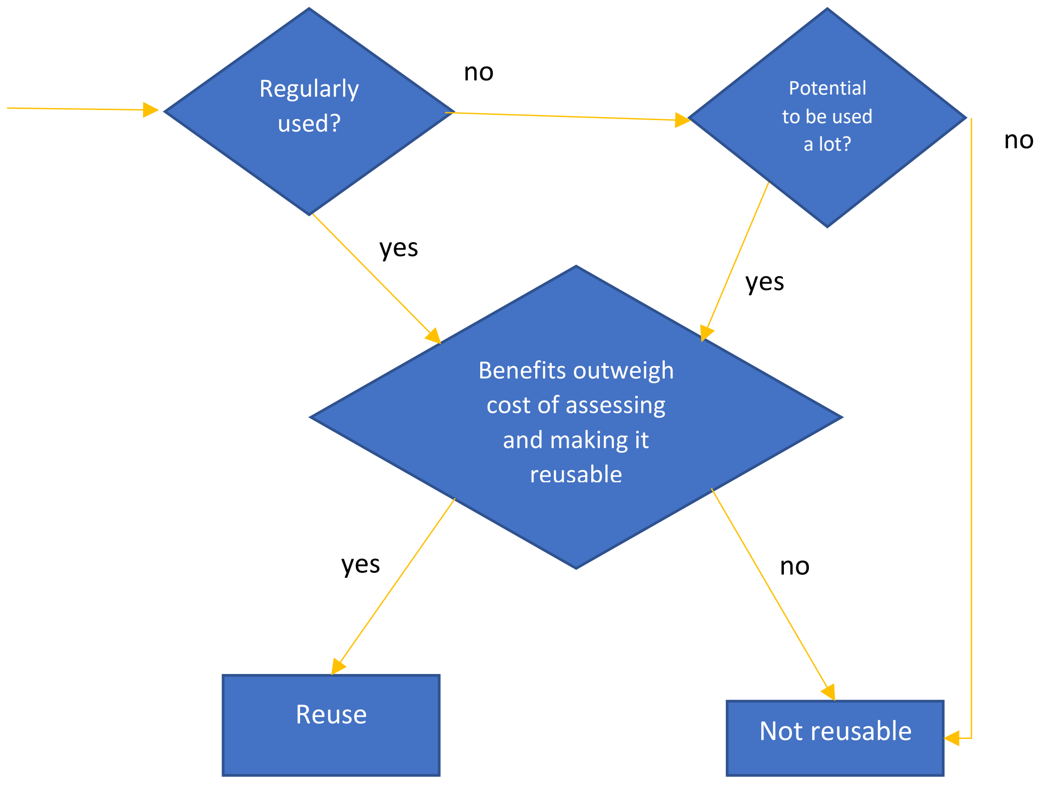
```
Def temp_check(advanced_thermometer):

        // for advanced thermometers, checks temperature and assess appropriateness.

        temp = int(advanced_thermometer.value)

        if temp > 200: out = (temp, “Really hot”)

        elif temp > 120: out = (temp, “hot”)

        elif temp > 30: out = (temp, “normal”)

        elif temp > -30: out = (temp, “cold”)
```

```
else: out = (temp, "really cold")
if out[1] in ["really hot", "really cold"]: // only want temp values not really cold or hot
        raise Exception // remove thermometer and other equipments
return out // appropriate
```

The method above, is very similar to part a, but the values has changed because now the equipment can withstand more extreme conditions. At first, temperature values of over 100 or below 30 is inappropriate but now with the new method, it can now be appropriate. Only values above 200 or below -30 is now deemed inappropriate.

c. The limitations of these modules with respect to ensuring perfect exception management is that it is based on the requirements and the context. For these to be reused, the requirements and context of the new and the old needs to be the exact same to ensure perfect exception management, assuming that the reusable module was tested appropriately. If the new requirements aren't the exact same, but very similar, the reusable modules might need to be changed to ensure it handles all test cases. Similar to part a and b, when the equipment and possible values changed, the module had to accommodate these changes. The reusability of a module is dependent on whether the new function has the exact same or very similar context or requirements. If it does, then the module can be reused either with no changes, or some changes to accommodate the difference in context and requirements.

2. Reuse Certification
    a. Flow diagram to assess reusability


Regularly used?
no
Potential to be used a lot?
no
yes
yes
Benefits outweigh cost of assessing and making it reusable
yes
no
Reuse
Not reusable

b. I think the only part you can automate in this process is assessing whether the code component is regularly used or have the potential to be used a lot. To do this, you can keep track how many times a component is used or whether it is a critical but general piece of component. Keeping track of how many times it is used is easy. As for determining whether a component is critical but quite general, the value to look for can be how many other libraries or pieces of code are using this component. If a variety of other components are using the same specific component, then it has the potential to be used a lot. These two things can be automatically processed.

The challenging part is determining whether the benefits of making the code reusable outweigh the costs of assessing and making it reusable. If the benefits outweigh the costs, then it should be made reusable and if not, then just leave it be. The automated process narrows down a list of possible reusable code components, but it doesn't instantly mean that it should be made reusable. A human must look and research the code and how the code can be reused. Specific context and requirements means possible changes to tests and code and it's hard for a computer to determine the reusability based on those. The computer can only determine the reusability of the code component based on quantifiable things, such as how many times its used or how many other components depend on this specific component. The hard part, whether the benefits outweigh the costs, is for humans to assess.