SYSC 4005 - Term Project

Deliverable 1

Group 58

Jacob Martin - 101000849
Kaj Hemmingsen-Beriault - 101010131

1 - Problem Formulation

A manufacturing company uses inspectors to ensure component qualities are satisfactory before sending them to a workbench to assemble products. There exist 3 components, titled C1, C2, C3; 3 products titled P1, P2, P3; 2 Inspectors which inspect the components, and 3 workbenches that assemble components into products.

The first inspector focuses solely on C1, while the second handles C2 and C3. Products 1, 2, and 3 require components C1, C1 and C2, and C1 and C3 respectively. These components must be present before a product can be assembled.

Each workbench can hold at most 2 of each component that it requires, while inspectors may supply them with an infinite amount of components. Inspector 1 handles the distribution of components based on which buffer is the smallest, giving highest priority to workbench 1, and lowest to workbench 3 in the event of a tie.

A simulation is needed to provide an estimated performance evaluation for this facility.

2 - Setting of Objectives and Overall Project Plan

The goals of this simulation are to find:

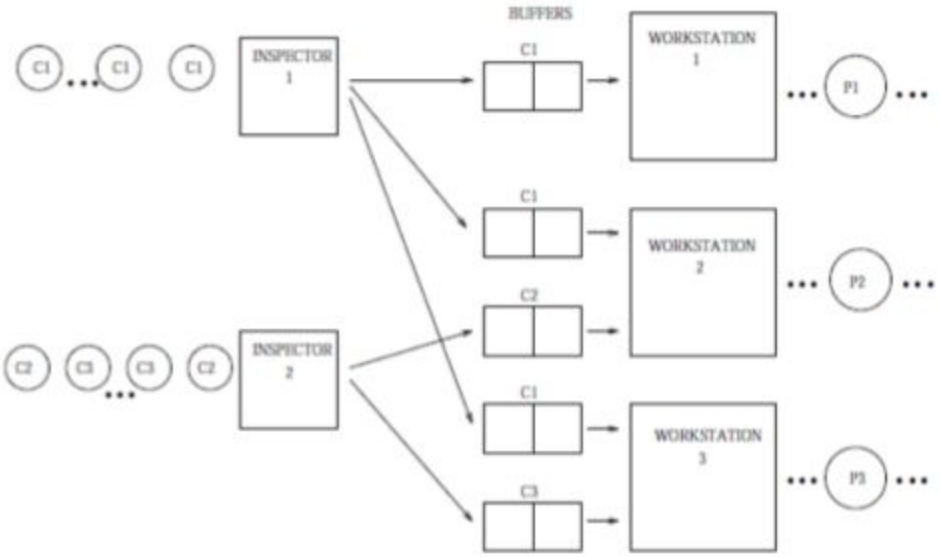
1. The throughput of the facility in terms of products made per unit of time.
2. The idle time of each inspector.
3. A means of reducing this idle time of each inspector and a new C1 distribution method to help increase throughput.

Due to sample dataset sizes, a simulation is best used rather than manual computation. The plan is to construct a program in golang that models each component, product, inspector, and workbench with respect to inspection times and assembly times. From there, repetitive runs will be done to optimize C1 distribution and reduce inspector idle time.

3 - Model Conceptualization

The model consists of four modules and one main class. There exist two entity classes, being the Component and Product classes. The Component has a name, id, and an associated read in file for data simulation. The Product has a list of required Components, and its own name. These are the base objects for Workbenches and

Inspectors to build on. The overall structure is very similar to that in the project description provided on CuLearn, used below.



The difference between this model and the planned one is that workstations handle the buffers themselves, and store product information within themselves. Inspectors then add components to a workstation's buffer.

4 - Model Translation

For our choice of modeling language we decided to go with golang. Golang is a statically typed, compiled programming language designed at Google. Golang is compiled to machine language and is executed directly unlike many other languages such as java who have to pass through the JVM first. Golang also has a special implementation for concurrency called "goroutines" which are an abstraction over a thread and allow for a new level of parallelism allowing for context to switch on the cpu faster allowing for faster multicore processing. Take into all of these considerations, golang is our choice of language to allow for faster computation times, in case we want to expand out models and or dataset and to use the built in concurrency to allow for a multithreaded solution.

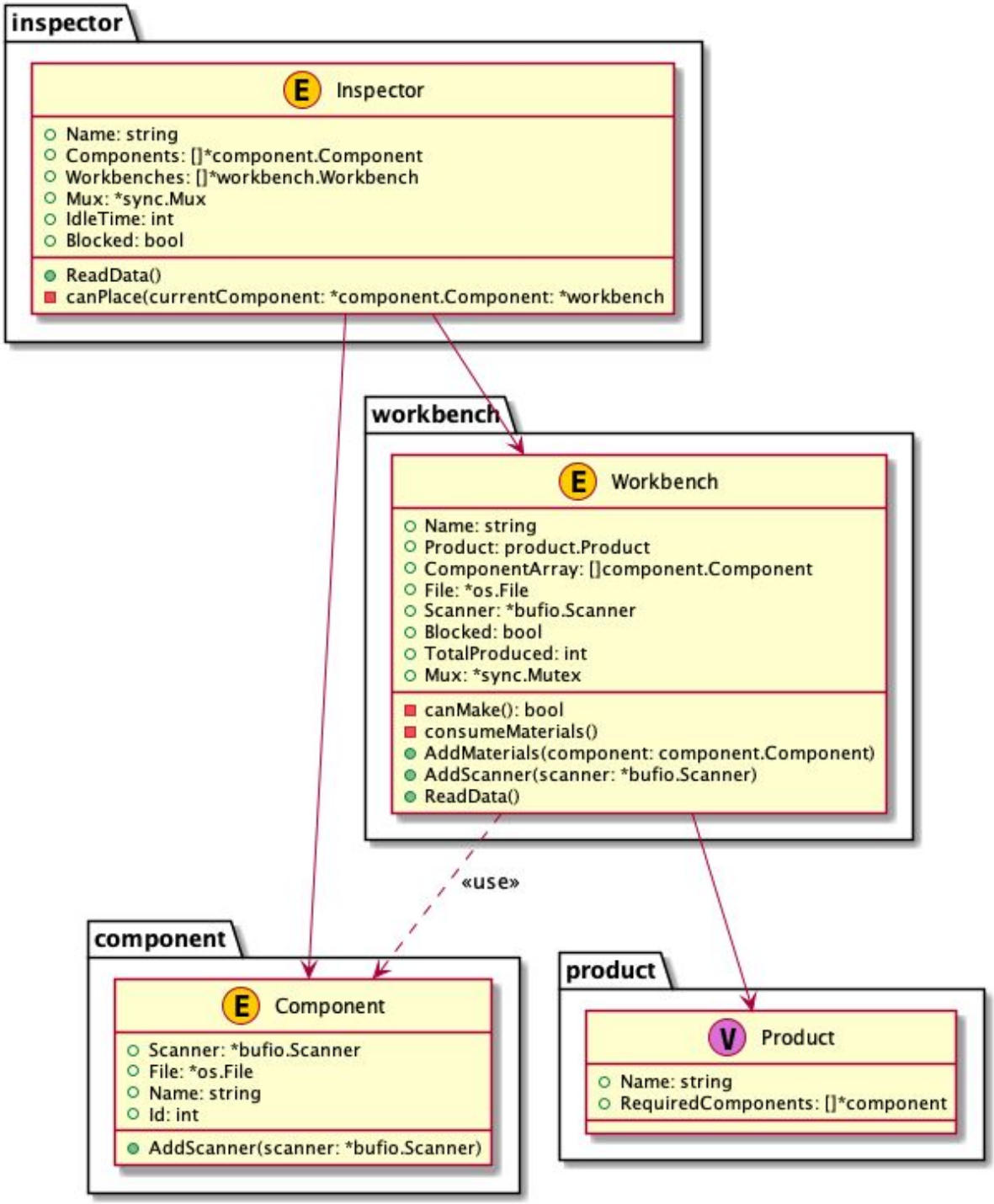


Figure 1: Class diagram

We are deciding on splitting up the required entities of the problem into their own classes. The resulting is; an inspector class, a workbench, a component and a product. With these entities we decided to use a top down approach where the inspector is in

charge of all entities below it. See figure 1 for class diagram. Each inspector is instantiated with a name, the components it can produce, the workbenches it can supply too and a mux which will act as our thread lock.

```
func NewInspector(name string, components []*component.Component, workbench []*workbench.Workbench, mux *sync.Mutex) *Inspector {
    return &Inspector{
        Name:        name,
        Components:  components,
        Workbenches: workbench,
        Mux:         mux,
    }
}
```

The workbench is made with a name, the required product. It makes the file it reads random data from and once again a mux that can act as a lock .

```
func NewWorkbench(name string, product *product.Product, file *os.File, mux *sync.Mutex) *Workbench {
    return &Workbench{
        Name:           name,
        Product:        product,
        ComponentArray: initComponentArray(product),
        File:           file,
        Mux:            mux,
    }
}
```

A component is made of a name, an id and the file with the random data that the component will need to produce.

```
func NewComponent(name string, id int, file *os.File) *Component {
    return &Component{
        Name: name,
        Id:   id,
        File: file,
    }
}
```

Finally a product is made of a name and an array of the required components.

```
func NewProduct(name string, requiredComponents []*component.Component) *Product {
    return &Product{
        Name:               name,
        RequiredComponents: requiredComponents,
    }

}
```

Each inspector and each workbench is spawned in their own thread to allow for independent calculations of each of the random wait times provided by the supplied files. Once each thread is running they will all begin to read the file data, in the case of the inspector with multiple components that it can produce it will select on from random. In terms of the workbench threads it will try to produce a component only if it has the required components in are in its buffer. When an inspector has completed a component ill will check all of the workbenches in its memory and try to place the component on its most suitable choice. If it can not place the component at that time it will be set to a blocking state until it can find a place to put the component. Once a workbench has completed a component it will consume materials in the buffer and once again try to complete a component. This cycle continues until all workbenches and inspectors are either finished or in a blocking state.