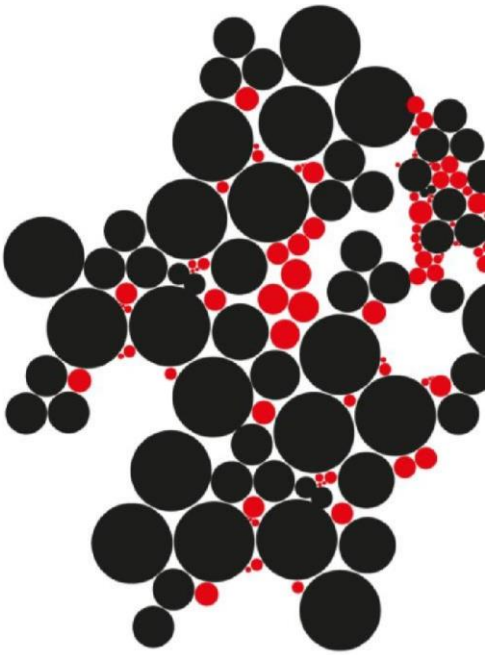
MASTER THESIS



The Modularity of the Resource Utilization Model in Python

Dennis Windhouwer

Department of Computer Science
University of Twente

Supervisors
prof.dr.ir. Mehmet Aksit
dr. Pim van den Broek
dr.ing. Christoph Bockisch
ir. Steven te Brinke

UNIVERSITY OF TWENTE.

DEC 2014

Contents

1. Introduction

1.1 Goals and Motivation

Achieving green software by reducing its energy consumption is becoming more and more important. One proposed method for achieving greener software is the resource utilization model (RUM) [9], which aims to extend the software with energy optimizers. But, as one of the main problems software is facing today is complexity [16], the impact of these energy optimizers on the complexity of the software must be minimized. To reduce the impact on the complexity of the software, the RUM method aims to modularize the energy optimizers from the rest of the software, separating the functional concerns from the optimization concerns.

The RUM style distinguishes among three types of components. Functional components, which each implement part of the functionality of the system. User components, which represent usage scenarios for the system. And optimizer components, which optimize the resource behavior. Functional and optimizer components need to have suitable interfaces to each other, so that the optimizer components can gather the information necessary to optimize the energy consumption, and so the optimizer component can take the necessary actions to optimize the energy consumption.

A functional component has a RUM, which represents the relations between the services provided by the component and the resources which it requires. This RUM has the form of a state transition diagram. A state transition diagram consists of one or more states and the transitions between them. The RUM's states are annotated with resource behavior, describing how many resources they consume and produce. Different states have different behaviors, switching between these states thus modifies the resource behavior of the system. Transitions between states can then occur after the invocation of one of a component's services. To ensure the separation of the functional concerns from the optimization concerns, the RUM of a functional component needs to be capable of intercepting these service invocations, allowing the RUM to execute state transitions automatically, when applicable.

An optimizer needs to analyze the states, the resource consumption, and provided services of a RUM in order to optimize the software and minimize the energy consumption. The optimizer also needs to introspect the state transitions of a RUM to determine which services it must invoke, so it can change the state of a RUM from its current state, to another more desired state. These optimizer components need to be interjected between other components at runtime. By interjecting the optimizers at runtime, the separation of the functional concerns from the optimization concerns can be ensured.

An implementation of the RUM style had not yet been realized, thus how modular a program could actually be when it was implemented in this style, and the impact of modularizing the energy optimizers from the rest of the software, was still unclear. An investigation into suitable programming mechanisms and languages, for the required functionality of a development environment for the RUM style, had already been done [18]. One of the investigated languages was Python [6]. Python is a dynamically typed languages, which offers many options for meta-programming [1], and offers a lot of options for introspection and intercession. Python was found to offer suitable mechanisms for a development environment for the RUM Style.

Our goal is to investigate how modular a program can be when it is implemented in the RUM style. In order to investigate this, we created a Python library for the RUM style, using the mechanics which were found suitable in the previous investigation. This library is capable of intercepting service invocations, so a RUM's state transitions can be executed automatically. Optimizer components can

also be interjected between any two components seamlessly, and the optimizers can determine how they can trigger desired state transitions in order to change the state of a RUM to a state with a more desired resource behavior.

In order to investigate the modularity of the RUM style, we first implemented an example program in the object-oriented style, using design patterns [11]. A functionally similar program was then also implemented in the RUM style, using the created library. These programs were then compared on their modularity.

1.2 Approach

To determine the modularity of the Resource Utilization Model in Python, we compared a program implemented in the RUM style, to a functionally similar program implemented in the object-oriented style with design patterns [11].

First we investigated how to measure the modularity of the resource utilization model. We identified several traits which we wanted to measure, and then investigated suitable metrics for measuring these traits.

Next, we implemented a program in the object-oriented style. This was a best effort at creating a modular object oriented program. By first creating this program, and ensuring it was as a modular as possible, influences from the RUM style on the program's design could be minimized.

Then we created the development environment, as the program in the RUM style cannot be implemented without it. The development environment was designed to be highly extensible, so that new functionally can easily be added to it, and so existing functionality can be extended.

Once the development environment was completed, we created a program in the RUM style, functionally similar to the earlier created object-oriented program. Work on this program continued until it too, was as modular as possible.

Finally, in order to determine the modularity of the Resource Utilization Model in Python, the two programs were compared to each other on their modularity.

2. Resource Utilization Model

This chapter discusses further details about the Resource Utilization Model. We explain what RUMs are, how they are created and how they are intended to be used. We use this to outline what the RUM library needs to be capable of in order to support the implementation of RUMs.

2.1 RUMs

A RUM represents the relations between the services provided by the components of a program, and the resources which these components require [9]. RUMs are expressed as state transition diagrams. Each RUM can have one or more states, each state contains information about the resource behavior of the component, which services it provides and requires, and which resources it provides and requires. The invocation of services can trigger a transition to another state, which can change the way in which the component consumes and provides resources. By using RUMs, the functional concerns are split from the optimization concerns. The functional concerns remain with the component, while the optimization concerns, such as resource consumption, are with the RUM. An optimizer can use the services of the component to trigger transitions to other states, allowing an optimizer to manipulate the resource consumption of the program.

To create RUMs, first the functional, user and optimizer components of the program must be identified. Each functional component implements part of the functionality of the program and interacts with other components to accomplish the program's overall function. The provided and required resources of each functional component should be modeled. Using a car and its components as an example, if a car's engine requires fuel, then this should be modelled as a required resource. The same must be done for the required and provided services. For example, one of the provided services of a car's engine is that it can be turned on.

Each user component represents a usage scenario. The most common usage scenarios should be modelled. The energy consumption of a system, and the effectiveness of an optimization strategy, can differ greatly depending on how the system is used. The functionality which is invoked during a usage scenario is modelled as required services. Based on the required services a RUM is designed, which describes how the services in question are used.

During the execution of the program, the optimizer components monitor and adapt the resource consumption of the functional components. Two main kinds of components can be identified. These are components which are directly controlled by the user components, and the functional components which directly consume the resource of which the optimizer wants to optimize the consumption. The optimizer must be interjected between these components, intercepting the interaction between them and applying its optimization strategy there. The provided services and types of resources of the optimizer must equal the required services and types of resources of the components directly controlled by the user components, while the required services and types of resources of the optimizer must equal the provided services and types of resources of the resource consuming components. The optimization logic must then be modeled as the component's RUM.

Figure 2.1 shows an example of a functional component with a RUM. This functional component has four services which it provides, these can be seen in the top right. They are to decrease and increase the throttle and to turn the engine on and off. The resource which it provides is rotations per minute (RPM), which can be seen in the top left. The resource which it consumes is Fuel, which can be seen in the bottom left. This component does not have any required services, but if it did then they would have been displayed in the bottom right. The component also has three different states, each with their own resource behavior and with transitions to other states.

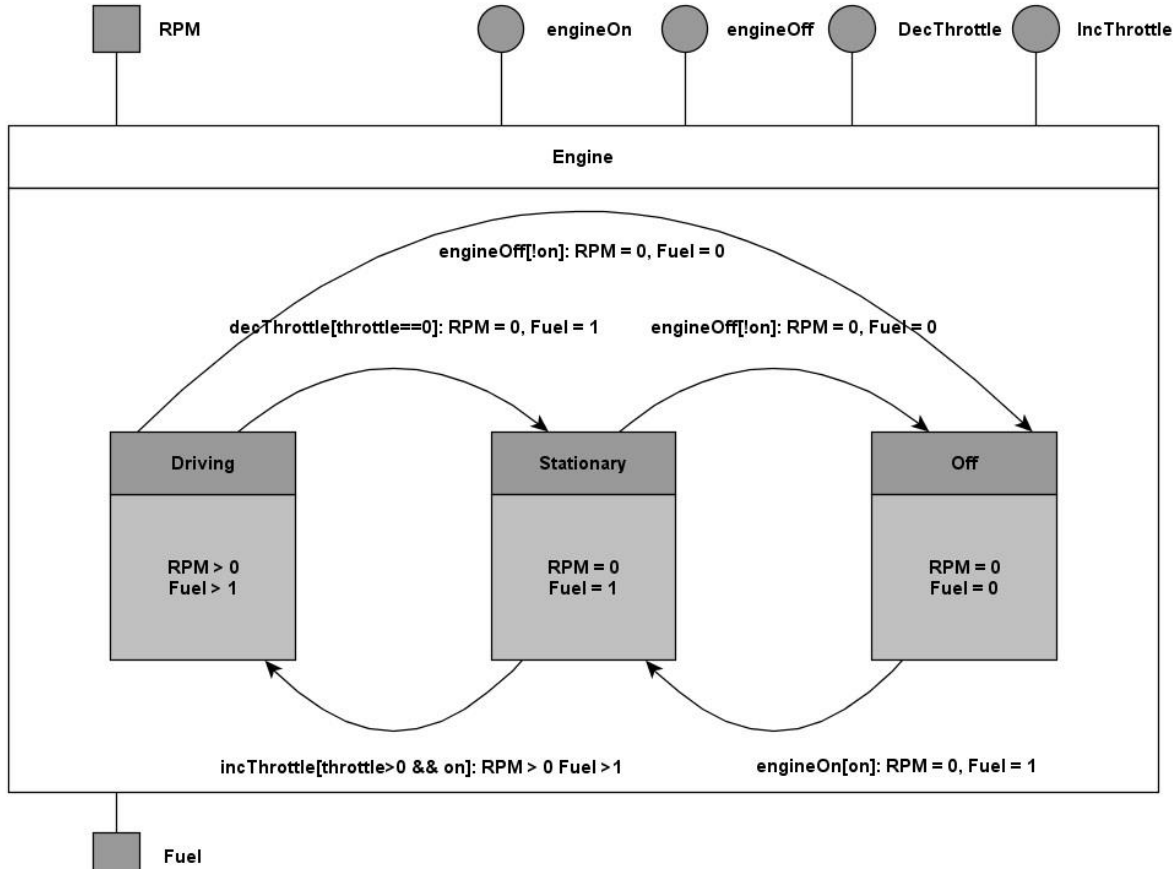


Figure 2.1

A user component can use the provided services in order to manipulate the engine. An optimizer component can then be interjected between the user component and the engine in order to manipulate the resource consumption and production of the engine. The following is an example of such an optimization. When the car has to stop in front of a railroad crossing, the optimizer can use information about average wait times at this crossing to determine whether the engine should remain in stationary mode, or whether it would be better for the fuel consumption to turn the engine off, and then back on once the crossing is clear. If it determines that it is better to keep the engine in stationary while the user is attempting to turn it off, then the optimizer would keep the engine running.

2.2 Requirements of RUM Library

The description of the RUMs and their workings shows several requirements which the RUM library will need to fulfill. As mentioned, service invocations can trigger state transitions, changing the state a RUM is in. But, as discussed in chapter one, one of the aims of the RUM method is to separate the functional concerns from the optimization concerns. If a component has to inform a RUM by calling a RUM's method every time a service invocation occurs, then this will couple the component to the RUM. This would prevent the separation of the functional concerns from the optimization concerns. Thus the RUM needs to be capable of intercepting relevant service invocations without help from the component to which the service belongs.

As mentioned, optimizer components need to be interjected between other components. These can for example be a user component and a functional component, or two functional components. In order to achieve the separation of the functional concerns from the optimization concerns, these components shouldn't make calls to the optimizer. In order to achieve the separation, the optimizer

should instead be interjected between the components without the components code reflecting this. Calls from the user component to the functional component should then automatically be redirected to the optimizer without either component being aware of it.

When an optimizer wants to modify the resource behavior of a component by changing the state of its RUM to a more desired state, then the optimizer needs to be capable of finding out which service invocations it must invoke, and in which order, so it can cause a transition to this more desired state. Information about these state transitions is contained in the RUM's state transition diagram. The states, and state transitions, thus need to be declared and stored in such a way that an optimizer can introspect them, allowing the optimizer to determine which services need to be invoked.

From these requirements, we conclude that the development environment should:

- Allow RUMs to intercept service invocations.
- Provide a way to declare introspectable states and state transitions for a RUM.
- Allow for the interjection of optimizer components between functional and user components.
- Allow for the optimizer to introspect a RUM's states and state transitions.

3. RUM Library

This chapter discusses the developed library for the Resource Utilization Model. In Section 2.2, several requirements for this library were defined;

1. A RUM needs to be capable of intercepting service invocations.
2. It needs to be possible to declare a RUM's state transitions in such a way, that they are introspectable.
3. It needs to allow for the interjection of optimizers between functional components.
4. It needs to allow for an optimizer to introspect a RUM's state transitions.

With the developed library, the focus was put on ensuring that it is extensible, and on ensuring that programs developed with the library could be modular. First we shall discuss the functional component's RUM and its main features; service invocation interception and its state machine. Then we shall do the same for the optimizer component's RUM.

3.1 Functional Component RUMs

As described in chapter two, each RUM is a state machine. This is also the case for the implemented Functional Component RUM, though it also contains functionality for in the interception of service invocations. The state machine depends on this functionality so the state machine can automatically update the current state of the RUM in response to a service invocation. This RUM thus contains two components, one which is the state machine itself, and another which is capable of intercepting service invocations, and then informs the state machine about which service was intercepted.

3.1.1 Service Invocation Interception

As mentioned in chapter two, the RUM has a state transition diagram. State transitions can occur when specific services are invoked. The RUM thus needs to update its current state after certain service invocations. The component to which the service belongs cannot inform the RUM about the service invocation, as the functional concerns need to remain separate from the optimization concerns. If the component were to inform the RUM directly, then this would couple it to the RUM and prevent the separation of those concerns. Thus the RUM needs to detect relevant service invocations on its own.

To this end we created the 'Interceptor' class. This class is responsible for intercepting service invocations. Each RUM inherits from the Interceptor class, giving the RUM access to the interception functionality. When a RUM informs the Interceptor that it wishes to intercept a specific method, then the corresponding method object is replaced by a method wrapper. As shown in figure 3.1.1.1, all the calls to the original method object will instead arrive at the wrapper. The wrapper then invokes the service which it wrapped, and after that it informs the RUM that the service was invoked. The wrapper also ensures that any value which the wrapped object returned is properly returned at the end.

The Interceptor class keeps track of which RUMs are intercepting which service invocations, and stores this information in a static dictionary. By default the Interceptor deploys a wrapper which, when invoked, first calls the wrapped method and then informs the Interceptor module that a service invocation was intercepted. The Interceptor module then checks which RUMs were interested in this service invocation, and informs these RUMs that the service invocation occurred. These RUMs then check if a state transition needs to be executed in their state machines, and if so, they execute the transition. The performance impact of this wrapper mainly depends on the amount

of state transitions which need to be checked by the RUMs which were interested in the invoked service, and on the amount of operators which these transitions contain. In the case of a single RUM, with a single state transition per state, each with a single operator, the performance impact was found to be around 8 microseconds.

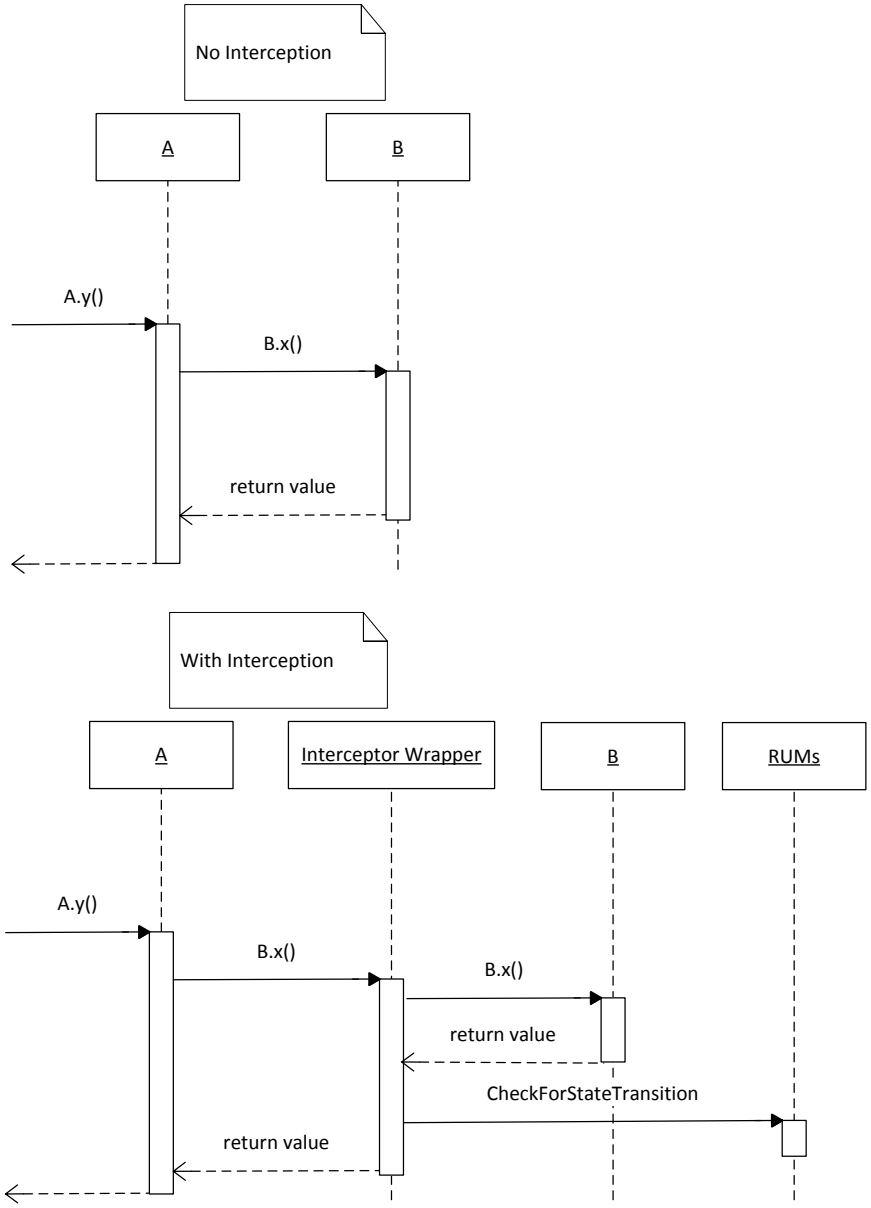


Figure 3.1.1.1 Interceptor wrapper sequence diagram

Another wrapper has also been implemented. When this wrapper is invoked, it first creates a dictionary containing the names and values of all the variables of the component to which the wrapped method belongs. Then it invokes the wrapped method. Once the wrapped method has returned, the wrapper again creates a dictionary containing the names and values of all the variables of the component. These two dictionaries thus contain all the values from before and from after the execution of the intercepted service. The wrapper then informs the Interceptor module about which service was invoked, and supplies the Interceptor module with these two dictionaries. The Interceptor module passes these two dictionaries on to the RUMs. These two dictionaries are passed on all the way down to the operands and operators of a state transition, allowing for the implementation of operands which, for example, represent the value of a field from before the

execution of the service, or operands which represent the amount by which a field was changed. Such operands have not been implemented. This wrapper will be used if a RUM is instantiated with the optional argument “lightweight = False”. The performance impact of this wrapper is much greater than that of the default wrapper. The more objects a component has in its dictionary, which includes both the variables and methods belonging to the component, the greater the performance impact, as the value of each of these objects is retrieved twice.

The current Interceptor does not support a precedence system between RUMs. If a specific RUM needs to check for, and execute, state transitions before another RUM, then this can only be done by having the first RUM declare the service invocations which it wishes to intercept, before the other RUM declares which it wants to intercept. But as all the information about which RUM is interested in which service invocations is currently stored in one place, it is possible to extend the Interceptor by also storing precedence related information. The wrapper then needs to be adapted so it takes this precedence into account when informing the RUMs about an invoked service.

3.1.2 State Machine

As mentioned in chapter two, each RUM has a state machine with one or more states and state transitions between these states. An optimizer sometimes needs to change the state of a RUM to a more desired state. In order for the optimizer to be able to do this, it needs to gather information about the states of the RUM, so it can determine which state is most desirable. Once it knows which of the RUM’s states is the most desirable, it needs to gather information about which state transitions need to be executed in order for the RUM’s state to change to desired state. With this information the optimizer can then invoke the necessary services, and cause the relevant transitions to be executed.

In order to gather this information about the states and their transitions, both need to be introspectable. In the RUM library, each RUM inherits from the ‘StateMachine’ class. This class manages the states, determines if a state transition needs to occur and executes a transition when necessary. The information on the possible state transitions is stored in the states themselves and not in the state machine. Each state is aware of which states can be reached from it, and which guards must be met before specific transitions may be executed. The state transitions are stored in a dictionary, where the guard is the key, while the next state is the value. This way it is possible to define multiple different transitions from state A to B, but no two transitions can have the exact same guard attached to them. The state machine itself does not directly know anything about the possible transitions, but it can retrieve this information from its states. The state machine contains a method called ‘getStateRoutesFromTo’, which uses the states and their transitions to construct a directed graph, where the states are the nodes and the transitions the edges, and then calculates and returns all possible paths from a given state to another given state. As long as an optimizer has access to a RUM with a statemachine, it can use this method in order to find out which transitions need to be executed in order to change the RUM from its current state to a more desired state. Figure 3.1.2.1 shows an example state diagram with three states and four transactions. State A has transitions to states B and C, state B has a transition back to state A and state C has a transition to state B. If the optimizer were to use the statemachine’s ‘getStateRoutesFromTo’ method to find out how to get from state A to B, then this method would return the following list of lists: [[A, B], [A, C, B]], where the values are the actual state objects. The optimizer can, for example take the first of these two lists, A->B, and introspect state A’s state transitions in order to find the transition to B. Or it could take the second list, find the transition to C, and then introspect C’s transitions in order to find the transition to B.

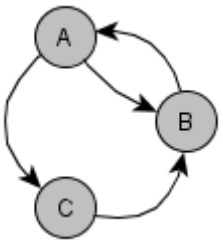


Figure 3.1.2.1 State diagram

In order for the optimizer to determine how it can cause a state transition, the guard part of the transition also needs to be introspectable. An earlier investigation into suitable programming mechanisms [18] showed that using classes to build an AST tree, is an introspectable and very extendable solution for defining the guard part of state transitions. Figure 3.1.2.2 shows an example of such an AST tree. This AST tree represents the following expression: (y+1)*(x-4)==(y+z). It contains a total of five binary operators. In the tree each operator, together with its two operands, is contained inside a Guard class. In this figure, the values of x, y and z represent FieldVariableOperands. The +, -, * and == signs represent Operator classes. Both the Guards, FieldVariableOperands and Operator classes are explained below.

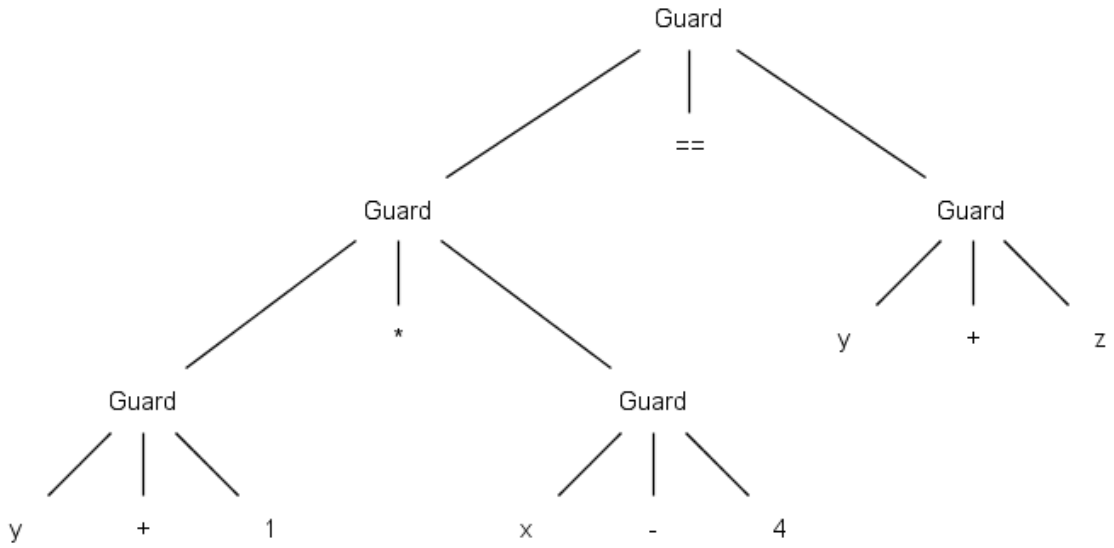


Figure 3.1.2.2 State transition

Three types of guards have been implemented, the basic guard has an operator and two operands, as shown in figure 3.1.2.2. There are also composite guards, the AndGuard and OrGuard, which can both contain multiple guard objects. The And, and Or operators have also been implemented as binary operators, but these operators can only work with two operands. The composite guards can have more than two, but they must be other guards. This allows for shorter but wider AST trees. The third guard is the guard for the invoked service. This guard's constructor takes a method object as an argument. This method object should be the service in question. The state machine checks if a state transitions needs to occur after a service has been invoked. When the state machine is informed about the service invocation, it is also informed about which service was invoked. When the invoked service guard is evaluated, it checks if the invoked service is the method object that was supplied to

it. This method object is the exact same object which belongs to the component, thus also sharing the same name. Through introspection the name of this method object can be retrieved, allowing an optimizer to learn which service it has to invoke.

The operators have been designed with extensibility in mind. Each operator contains an 'op' variable. This variable should be a function object, and is automatically executed when a guard is evaluated to determine if a transition should occur. By using a function object, it is possible to smoothly create and define new, and more complex, operators. Each operator class also has a 'description' variable, which can be treated as its name or identifier. When introspecting an operator, this descriptor can be used to learn exactly which operator is being used. Code block 3.1.2.3 shows the implementation of the 'not equals' operator. As can be seen, this operator only contains a constructor, and inside this constructor it calls the constructor of its superclass, supplying it with two extra arguments, the built-in Python 'not equals' operator, and "!=" as the description. When introspecting operators, this description can be used to identify an operator. Another option for identifying an operator is to use Python's built-in *isinstance()* function in order to determine the operator's class.

```
1  class NotEqualsOperator(BooleanOperator):
2      def __init__(self):
3          BooleanOperator.__init__(self, operator.ne, "!=")
```

Figure 3.1.2.3 Not Equals Operator

Finally there are the operands. Any base value can be used for an operand, but there are also two special operand classes for more complex cases. The first of these is the 'FieldVariableOperand'. This operand allows for the usage of fields of instances as operands. In figure 3.1.2.2 this operand has been used for three different fields, *x*, *y* and *z*. Normally, when the field of an instance is supplied when constructing an operand, the value of this field is simply passed. For the FieldVariableOperand to supply the current value of a field, the operand must have a reference to the component and it must know the name of the field. The operand can then use Python's built-in getattr() function to retrieve the current value of the field. The FieldVariableOperand thus returns the current value of its assigned field whenever the operand is used, ensuring that the value always reflects the actual value of the field. The FieldVariableOperand can be instantiated with an optional argument, *step*. When not supplied, the default value of *step* will be one. This argument is used by the Query system, and is described further in section 3.2.2.

The second of these operands is the 'FunctionValueOperand'. This operand can be used to add a function from an instance, for example a getter method, as an operand. This operand executes the function and returns the value returned by the function whenever the operand is used. If the used function causes side-effects then this can have unexpected results, so caution should be taken when using this operand. New operands like these two can be added, by extending the AbstractOperand class. As mentioned in section 3.1.1, other operands can also be implemented, for example an operand which returns by how much a specific field was modified by the invoked service, but such operands do require that the second, more performance heavy, wrapper is used.

When an optimizer needs to know how to cause the execution of a state transition, it can introspect the guard class and access its operator and operand. By introspecting the AST tree it can gather all the FieldVariableOperands used in the tree. Inside each of these operands the component to which the field belongs, and the name of the field, is stored. This information can be used to determine which fields of which components are involved. An optimizer can also introspect the guards in order to access the operators. It can then check the description of an operator in order to find out which operators have been used in the state transition, or it could use Python's built-in isinstance() function

in order to determine the class of the operator. The optimizer can also introspect the guard for the invoked service, if there is any, and determine which service must be invoked. With this information, all aspects of the transition can be identified, it is then possible to determine which combination of values for the fields would result in the guard returning true when it is evaluated, and which service must be invoked in order to cause the transition to another state.

3.2 Optimizer Component RUMs

3.2.1 Optimizer Interjection

As mentioned in chapter two, the library needs to allow for the optimizer to be interjected between two components, while ensuring the separation of the functional concerns from the optimization concerns. To ensure this, the two components should not be aware that the optimizer has been interjected between them. This rules out design patterns such as the strategy pattern, as in that pattern the component would know about the strategy and forward calls to it on its own. The functional components should not have to be rewritten in order to enable the interjection of the optimizer. Thus a mechanic is needed, which can interject an optimizer between two components, without having to modify either of the components.

The technique used with the interception of service invocations is applicable here. If optimizer C needs to be interjected between components A and B, so that all calls from A to B, but not from B to A, are redirected to C, then all the methods in component B can be wrapped by a method wrapper. This method wrapper then needs to determine where the call to its wrapped method came from. Did it come from A, or from another component? If the call came from A, then the method wrapper can redirect the call to C, otherwise it can let the call continue to B.

In Python, it is possible to introspect the current frame object [4]. Frame objects represent execution frames. The current frame object contains information about the method which is currently being executed. An example of several of the attributes of such a frame objects are: ‘f_back’, which is the frame object which called this frame, or None if this is the bottom stack frame; ‘f_code’ is the code object which is being executed in this frame, this can be used to, for example, retrieve information about the raw compiled bytecode; f_locals, which is the dictionary which is used to look up local variables, like the arguments supplied to the method which is currently being executed. The methods available for retrieving the current frame were found to come at a minimal performance cost of around 0.2 microseconds.

Take code block 3.2.1.1 as an example. This code block contains a main method and two classes: Component and OtherComponent. OtherComponent has a variable *i*, Component also has a variable named *i*, and a variable *B*. In the main method, one instance of each of these classes is created, and variable B represents the instance of the OtherComponent class. Here method x from the Component instance calls method y from the OtherComponent instance. If we are currently executing method y, then the current frame object contains information about method y. The f_locals attribute of the current frame can then be used to access the arguments supplied to method y: self and amount. As mentioned earlier, in Python the first argument of each non-static method (usually called self) represents the current instance, in the case of method y the first argument is the instance of the OtherComponent class. The f_back attribute of the current frame is the frame object which called the current frame object, in this example the f_back attribute thus represents the frame object which represents method x from the instance of the Component class. The f_locals attribute of this frame object can then be used to access the instance of class Component, from which the call to method y was made, in this case A. This enables us to detect from which instance a call originated.

```
1  def main():
2      B = OtherComponent(5)
3      A = Component(6, B)
4      A.x(5)
5
6  class Component(object):
7
8      def __init__(self, i, B):
9          self.i = i
10         self.B = B
11
12     def x(self, amount):
13         self.i = amount
14         self.B.y(amount)
15
16
17 class OtherComponent(object):
18
19     def __init__(self, i):
20         self.i = i
21
22     def y(self, amount):
23         self.i += amount
24     def z(self):
25         self.i *= 2
```

Code block 3.2.1.1

The component responsible for interjecting optimizers has been called the ‘Interjector’. This component uses method wrappers as described above. When it is asked to interject an optimizer between instances A and B, it wraps all the methods declared in B’s class and superclasses, excluding build-in methods from Python. The Interjector also stores information about which methods it has wrapped. This information includes to which optimizer a call must be redirected, and from which instance(s) the call should have come in order to warrant redirection. This information is stored statically in a dictionary. The Method wrapper can use the current frame’s f_back attribute to find out where the call to the method wrapper originated from, and then the wrapper can check the stored information to confirm if this instance is one from which calls need to be redirected. For example, using code block 3.2.1.1, assuming that an optimizer was interjected between instances A and B, so that all calls from A to B should be redirected to an optimizer, then B.y() will be wrapped by a method wrapper. If B.y() is called, then this call arrives at the method wrapper, the method wrapper then uses the current frame’s f_back attribute to get the frame of the method which called the wrapper. The wrapper can then determine if it was called by a method from instance A, or by a method from another instance. As shown in figure 3.2.1.2, if the wrapper determines that it needs to redirect the call, then it redirects the call to the optimizer, the optimizer then optimizes the instance of class B and in this case determines that it should invoke method z instead of x. If the optimizer determines that it should not redirect the call then it calls the wrapped method, B.x().

Instead of interjecting an optimizer between components A and B, redirection the calls to all of B’s methods, we have also made it possible to interject the optimizer between a component and a method of another component, for example between component A and B.foo(). This way, only the calls to desired methods are redirected to the optimizer, instead of the calls to all methods. If only the calls to a select few methods should be redirected, then the optimizer should be interjected between a component and a method instead of interjecting it between two components. This way, the amount of deployed method wrappers is limited, thus also limiting the size of the dictionary in

which the information about deployed wrappers is stored. This limits the time required to find out if an intercepted call needs to be redirected or not, thus keeping the performance impact to a minimum.

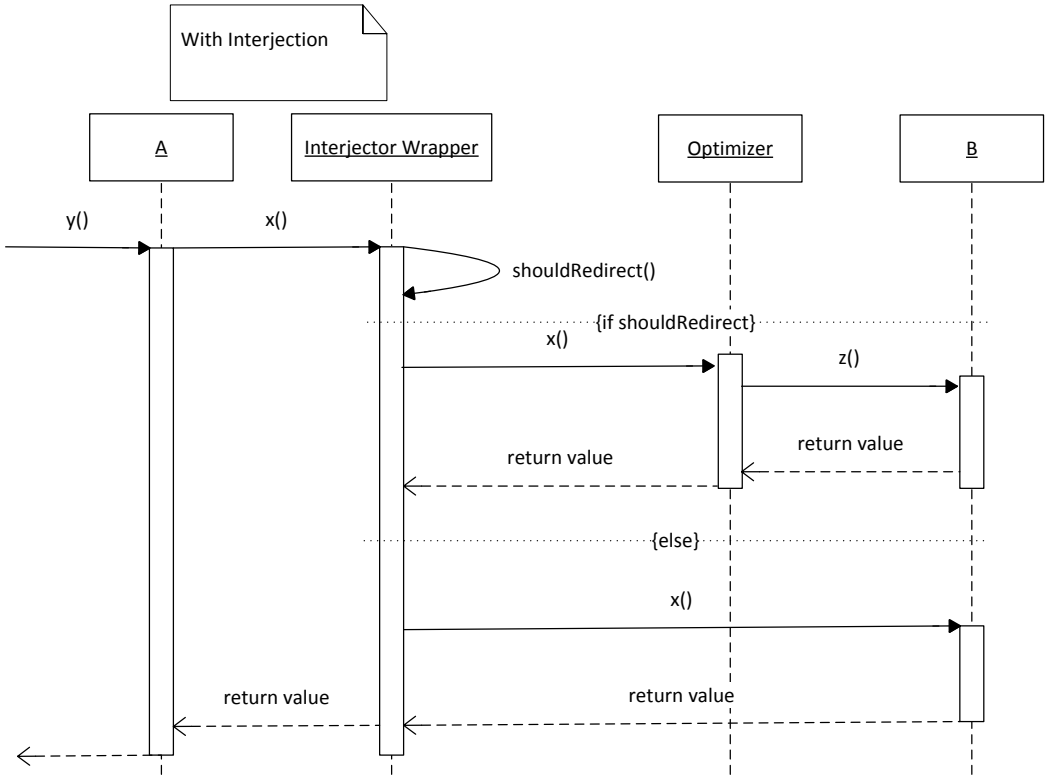


Figure 3.2.1.2 Interjector wrapper sequence diagram

The existence of the Service Invocation Interceptor complicates matters slightly, as it too wraps methods in method wrappers. For example take the following scenario: the Interjector interjected an optimizer between instances A and B, B.y() has been wrapped by the Interjector. After this a RUM desired to intercept the service invocation of B.y(), and used the Interceptor module to wrap B.y(). This Interceptor didn't actually wrap B.y(), as this method was already wrapped by the Interjector, so instead the Interjector's wrapper was wrapped. If A.x() were now to call B.y(), then this call will first arrive at the Interceptors wrapper, this wrapper then calls the Interjector's wrapper. When the Interjector checks where the call came from, it will determine that the previous frame does not belong to A.x(), while the original call did originate from there. In order to handle this situation, the Interjector's method wrapper checks if the previous frame belongs to a method wrapper from the Interceptor module. If this is the case then it uses this frame's f_back attribute to go back one extra frame, and then uses this frame to determine where the call actually originated from.

This can also occur in the opposite order, the Interceptor could first wrap a method and then the Interjector can also wrap it, effectively wrapping the Interceptor's wrapper. Now if A.x() were to call B.y() then the Interjector's wrapper would detect that the call came from A, and redirect the call to the optimizer. The Interceptor's wrapper would thus never be called, which is as expected as the wrapped service also wouldn't be invoked if the Interceptor hadn't wrapped this service. If the call to B.y() did not originate from A then the Interjector's wrapper would invoke it's wrapped method, which is the Interceptor's wrapper. The wrapper would then function as expected and invoke the wrapped service. Thus this order of interaction between the two wrappers does not create any issues

and both wrappers will function as expected.Undo functionality has also been added, if an optimizer was interjected, but needs to be replaced by another optimizer, then the interjection of the old optimizer can be undone. It is also possible to interject an optimizer between two components, and then to use this undo functionality to exclude one or more methods.

Some limitations do exist with this implementation. In Python all methods and functions are objects. It is possible to replace an instance's method with another method. If this is done, then a wrapped method could be replaced by a new, unwrapped, method. Calls to the new method will not be redirected automatically, instead the optimizer will need to be interjected again, even if you choose to deploy the optimizer between two components and had all its methods wrapped.

It is possible to overcome this limitation, as all assignments, including changing an instance's method, are done through Python's built-in setattr() function. Wrapping this function inside a method wrapper should make it possible for the Interjector to detect that the program is trying to replace a wrapped method with an unwrapped method. The Interjector could then be informed, and respond by wrapping the unwrapped method after the completion of the setattr function.

Another limitation is how the Interjector determines the origin of a call. It does this by checking from which instance the call originated. The first argument of a method normally represents the current instance, but this is not the case with static methods, instead their first argument can be anything. If optimizer C has been interjected between components A and B, and a static method from A calls a method in B, then the Interjector cannot determine where the call came from, and will never redirect it to the optimizer. The Interjector cannot currently determine if the calling method is static or not, thus if a static method from class D has as its first argument component A, and calls a method in component B, then this call will be redirected when it shouldn't be. In Python it is possible to detect if a method is a static method or not, as static methods are of the type 'function', while other methods are of the type 'instancemethod', but it is not possible to obtain the actual method object from a frame. What can be obtained is a code object, which contains the compiled bytecode of the method object, but the method object itself is not accessible from there. Thus the current Interjector cannot determine whether the calling method is static or not. Other ways to determine whether the calling method is static or not might exist.

One last limitation is that this wrapper uses frame objects. These frame objects are retrieved through the built-in sys._getframe() function. This function relies on Python stack frame support in the Interpreter, which isn't guaranteed to exist in all implementations of Python. If an implementation without Python stack frame support is used, then the function will likely return None, or a dummy frame object, and then the Interjector won't function. We do not know of a way around this limitation, as frames are necessary to access the first argument of the calling method, and to then determine from which instance the call originated.

3.2.2 Query Capabilities

As mentioned in section 2.2, an optimizer needs to be able to introspect a RUMs state transitions, so it can determine which state the optimizer wants the RUM to be in, and so the optimizer can determine which state transitions it must trigger in order to achieve this. In 3.1.2 we discussed how the states and their state transitions are defined to be introspectable. In order for an optimizer to access this information, the optimizer's RUM has been given a group of getter methods which, as their first argument, require a RUM with a statemachine. Each getter method then returns specific information about this statemachine. An example of such a getter method is the 'getPossibleNextStates' method, which returns a list of all the states which are directly reachable from the current state of the given RUM. The optimizer component's RUM class can be extended

further with more getter methods if information is required which no getter method currently supplies.

An example query system has also been implemented, which determines which service invocations are required to reach a specific state, and which fields of which components must have which values in order for a guard to be evaluated to true. As an example, take figure 3.1.2.2 as the guard of the transition which needs to be executed in order to reach a desired state. The guard for this transition is: *(y+1)*(x-4)==(y+z)*. This guard does not contain any mention of a service invocation, which means that any service invocation to which the RUM is listening can potentially trigger the transition. Assuming that the current values of *x*, *y* and *z* are six, two and ten respectively, then the guard currently evaluates to 6==12, which is false. In order for the outcome to be true, *y* could be increased by six so both sides of the equals operator resolve to eighteen. The *x* variable could also be increased by two, or the *z* variable could be decreased by six. Combinations of modifications to multiple variables are also possible, for example: *x* could be increased by one, *y* increased by one and *z* decreased by one. Both sides of the equation then resolve to twelve. The example query system we have created returns a limited amount of possible combinations of actions, the optimizer then needs to execute the actions described in one of these combinations so the transitions can occur.

In code block 3.2.2.1, we give a simplified description of the algorithm which we use to determine some of the possible solutions for state transitions. When a state transition is queried, like the one from figure 3.1.2.2, then first the topmost guard is examined. In this case, that is the guard with the equals operator. The first step is to examine the two operands, if these operands are also guards then they must be examined first, starting with the left guard. Thus before the guard with the equals operator is fully examined, first the guard with the multiplication operator is examined, and before that guard is examined first the guards for the *y+1* expression, and the *x-4* expression are examined. When examining the guard for the *y+1* expression, neither operands are guards, so next the operator is examined. When examining the operator the algorithm checks if the operator returns a Boolean result or not, it does this by evaluating the operator, similarly to how it is evaluated when determining if a state transition needs to occur. If it does not return a Boolean value, like with the *y+1* operator, then a list of all the fields used by this guard is created and returned. In this case that is only the *y* operator. Thus when examined, the *y+1* guard returns a description of the field *y*, and the *x-4* guard returns a description of the field *x*. The multiplication guard returns both of these field names to the equals guard. This brings us back to the equals guard, where the results of examining its two operands are two lists of fields, for the left operand these are *y* and *x*, for the right operand these are *y* and *z*. It then examines the operator of the topmost guard. This operator does return a Boolean value when examined. Since it returns a Boolean value, the algorithm then checks if the two operands belonging to this guard also currently have Boolean values. In this case they aren't Booleans but numbers, thus the algorithm in code block 3.2.2.1 is used in order to determine the possible solutions for which this operator will return true. This algorithm has been simplified here, so a lot of the arguments used by the methods have been left out. Before the __determineQueryResultsHelper__ method is called, first one list of all involved variables is created, in this case the variables are *x, y* and *z*. Then the gap, or difference, between the left and right operands is calculated. In this example *x* has the value of six, *y* has the value of two and *z* has the value of 10, thus the left operand has a total value of six and the right operand a total value of 12. The gap is thus six. Then the __determineQueryResultsHelper__ goes through all the variables which have been supplied to it. For each variable it tries to find a solution by increasing or decreasing the value of the variable. Each variable in line 3 is actually a tuple, containing the name of the field, the component to which it belongs and the step size by which it should be changed. The __tryFindSolution__ method is called in order to find a solution, if one exists. This method contains a

while loop. In this while loop it first increases the value of the given variable at line 11. The amount by which it increases this variable depends on the given direction, and the value of the *step* variable belonging to the FieldVariableOperand. If this optional argument was not supplied when the FieldVariableOperand was instantiated then it defaults to one. In our earlier example where the value of variable *x* was six, it would thus increase *x* to seven or decrease *x* to five. The algorithm does not actually modify the field belonging to the component itself unless one of the operands used in the guard is a FunctionVariableOperand, as the value returned by the FunctionValueOperand might depend on the actual fields inside the component. Thus the usage of the FunctionVariableOperand can cause side-effects for multi-threaded programs, as in that case the component itself is actually being modified during the query process.

```
1       def __determineQueryResultsHelper__(self, fieldVariables, used, gap):
2           result = []
3           for variable in fieldVariables:
4               if not variable in used:
5                   self.__tryFindSolution__(variable, 1, gap, used)
6                   self.__tryFindSolution__(variable, -1, gap, used)
7           return result
8
9       def __tryFindSolution__(self, variable, direction, gap, used):
10          while True:
11              self.__adjustValue__(variable, direction *variable[2])
12              newGap = self.__determineGap__()
13              if self.__correctSolution__():
14                  result.append(self.createResult())
15                  break
16              else:
17                  used.append(variable)
18                  result.extend(self.__determineQueryResultsHelper__())
19              if not self.__correctDirection__(gap, newGap):
20                  break
21          self.__resetValue__(variable)
```

Code block 3.2.2.1 simplified query algorithm

Assuming that the algorithm increased x by one to seven. It then checks what the difference between the left and right hand sides of the guard now equates to, which is three, as the left hand now resolves to nine and the right hand still resolves to twelve. Then on line 13 it checks if the guard now resolves to true or not. If it does then we have found a correct solution and it calls a method which instantiates a new QueryResult class, which contains all the details about this solution. If we have not found a solution then it adds the current variable to the list of used variables, and then calls the __determineQueryResultsHelper__ method. Since a solution has not been found, this method is called and it now increases the value of variable *y* by one. The gap is now one, as the left hand side resolves to twelve and the right hand side resolves to thirteen. Again we still haven't found the solution, so it now increases variable z by one, which increase the gap to two. It now has no further variables to modify, so we arrive at line 19. Here it checks if further increasing the value of variable *z* can bring us closer to a solution or not. In this case the gap has widened from one to two, thus further increasing z will likely only widen the gap further and not bring us closer to a possible solution. Thus it breaks out of the while loop and resets the value of *z* back to ten. Then it arrives at line 6 and attempts to decrease the value of *z*. Decreasing the value of *z* to nine results in the guard resolving to true, so a solution has been found and it adds this solution to the result set, then breaks out of the loop and resets the value of *z*, returning to further attempting to modify *y*. It won't break out of the while loop in which it is modifying variable *y*, until either the gap starts to increase instead

of decrease, or until the gap overshoots zero and the gap goes from positive to negative. In this manner the algorithm will find a total of 28 solutions for this particular guard, far more than we need. If more variables are used in a guard then it will likely find even more solutions, which means that the more complex a guard is, the greater the performance cost is for finding a solution, and as this example with three variables already return 28 solutions, the scaling in this example is worse than quadratic, and this is likely also the case in general.

In order to limit the performance impact, the example query system's constructor accepts an optional argument (singleResult = True), which can be used to limit the returned results to just one result. If this argument is supplied then the query system will stop looking for solutions the moment it has found the first. This greatly limits the performance costs, but since the query system does not know how an optimizer can modify the variables of a component, it might not return a result which is useful for the optimizer.

If in the above example the equal guard instead had Boolean operands on both sides, then a different algorithm would have been used to determine the solutions. The algorithm for determining the solution for an operator with Boolean operands is much simpler. First it checks when this operator evaluates to true and when it doesn't. Since these operators are binary operators, they only have two operands which can resolve to either true or false, thus there are four possible cases in which the operator can return true. If one of the two operands was the guard from figure 3.1.2.2, then this guard would already have been evaluated and would have returned one or more solutions.

```
1  OrQueryResult containing:
2      AndQueryResult containing:
3          QueryResult (increase x by 1)
4          QueryResult (increase y by 1)
5          QueryResult(decrease z by 1)
6      AndQueryResult containing:
7          QueryResult (increase x by 2)
8          QueryResult (increase y by 1)
9          QueryResult (increase z by 3)
10     QueryResult (increase x by 2)
11     QueryResult (increase y by 6)
12     QueryResult (decrease z by 6)
13 And 23 other AndQueryResults
```

Figure 3.2.2.2 Example query system result

Thus this operand evaluates to true when one of these solutions is used, and it returns false when none of these solutions are used, which is the inverse of these solutions. For example if the operator in question is an equals operator then two results will be returned, one for which both the left and right hand operators resolve to false, and one for which both operators resolve to true.

The result shown in figure 3.2.2.2 are the 28 results which the algorithm in Code block 3.2.2.1 will return if the option to only receive one result hasn't been used. The result has an OrQueryResult class at the top, containing 28 QueryResults. Since there is an OrQueryResult at the top, only one of the 28 subresults needs to be executed in order to trigger the desired state transition. The first of these is the AndQueryResult in line 2. In order to execute this AndQueryResult, all its subresults also need to be executed. The result returned by this example query system needs to be parsed by an optimizer. Out of the 28 possible options in the OrQueryResult, one then needs to be chosen by the optimizer. The optimizer then needs to call services of the component in order to change the values of the three fields. The service called last should be one which the RUM, whose state we wish to change, is intercepting. Once this last service has been invoked, the RUM's state machine will be

informed and the transition to the desired state will be executed as the desired transition now evaluates to true. This example system shows that the state machine, its states and their state transitions are all introspectable enough to supply the optimizer with the information it needs in order to trigger a transition to a more desired state. This example query system does fall short if a Boolean operator is added which accepts one Boolean operand and one non-Boolean operand. If such an operator is added then the query system's __*visitBooleanOperator*__ method should be overridden, the operator can then be identified by its description or its class and handled accordingly.

4. Example programs

This chapter discusses the two example programs, explaining their general structure and functionality. Both the Object Oriented Program and the RUM style Program are functionally similar and structure wise have a lot in common, so what they have in common will be discussed first. Then the parts which they do not have in common will be discussed, first for the Object Oriented program and then for the RUM style program.

4.1 General Structure

Both programs simulate a delivery vehicle, which drives around a world, picking up packages and bringing them to their destinations. The programs are split into three parts, the world, the vehicle, and a controller class which can tell the vehicle what to do.

4.1.1 The world

The world is made up out of three parts, these are locations, the roads which connect these locations to each other, and a simple weather simulation system. The locations and roads together form a road network and have the form of an undirected graph, where each location is a node and each road is an edge between two nodes. The roads have several attributes, they can have a steepness, a maximum speed which a vehicle may not exceed, and similarly a minimum speed. The steepness of a road affects a vehicle's speed, going downhill is easier than going uphill.

Each location can contain buildings. These buildings serve as the pickup and drop-off points for the packages which the vehicle has to move. Some of these buildings can also function as refuel stations, allowing the vehicle to refuel its fuel tank. The packages have a starting location and a destination location, both of these are buildings. Each package also knows where it currently is, which is either in a building of some kind, or in a vehicle. Each package also has a weight and volume.

The weather simulation determines from which direction the wind is coming. This direction is always in relation to the vehicle, either coming from the front, the side or the back. The wind also has a strength level, the stronger it is the greater the effects of the wind are on the vehicle. Wind coming from the front will make it harder for the vehicle to get to its destination, while wind coming from behind will make it easier.

A WorldFactory class is included, which generates a default world with a big network, and then spreads several packages randomly throughout this world.

4.1.2 The vehicle

The vehicle is made up out of several parts. It has a propulsion, which is the combination of an engine and throttle, it also has a gearbox, a fuel tank, a trunk and a route planner.

The propulsion and gearbox together have a large impact on the speed of the vehicle. The propulsion has a throttle, which can have a value from zero to a hundred. The higher this throttle, the more fuel is consumed and the more rounds per minute (RPM) the engine generates. Several other factors also affect the generation of RPM by the propulsion, for example the steepness of the road which the vehicle is currently driving on, the weather and the weight of the packages which the vehicle is transporting.

The generated RPM is then further modified by the gearbox, it is divided by the ratio of the differential ratio and by the ratio of the gear itself. The ratio of the gear itself depends on the current gear, changing to a different gear will change this ratio, but it will not affect the differential ratio.

Finally the circumference of the wheels of the vehicle are taken into account, resulting in the current speed of the vehicle.

In both programs, classes are in place whose purpose it is to optimize the propulsion and gearbox, in order to either keep fuel consumption as low as possible, or to maximize the speed of the vehicle and get it to a destination as fast as possible.

The trunk is where all the packages are stored while they are in transit. It has a maximum volume, so there is a limit to the amount of packages which the vehicle can have in its trunk at the same time. The trunk also has functionality which returns the total weight of all the packages, this affects the fuel consumption of the vehicle. A heavier vehicle takes more energy to move than a lighter one.

The route planner contains functionality which can be used to find the fastest or shortest route from one location to another. It also knows the current location of the vehicle and has access to the current locations of all packages, allowing it to for example determine which package is nearest to the vehicle.

4.1.3 Controller

The controller is responsible for turning user input into actions. It has functionality through which the current throttle of the propulsion and the current gear of the gearbox can be modified. It is also possible to refuel the vehicle, and to load and unload packages at the current location. It also contains methods which take a package as their argument, and then either drive the vehicle to the location of the package and load it onto the vehicle, or drive the vehicle to the destination of the package and then unload it there. Finally, it contains a method which takes a location as an argument and then drives the vehicle to that location.

4.2 Object-Oriented Program

The object oriented program uses the strategy pattern to handle the optimization concerns. Both the gearbox and the propulsion have a strategy instance. The methods inside the gearbox and propulsion which were responsible for modifying the gear and throttle now inform their strategy instance when they are invoked.

Three different strategies exist for each component. The first is the manual strategy, which contains no optimization. If the controller tells the vehicle to increase its gear from four to five, then this is exactly what will happen.

The second strategy is the efficient strategy. This strategy contains optimization and aims to limit the fuel consumption, preferring to spend more time but less fuel in order to get the vehicle to a destination. If the controller tells the vehicle to increase its gear from four to five, then this strategy will only let that happen if the fuel efficiency of the vehicle would be improved. If the strategy determines that further increasing the gear to six would improve fuel efficiency even further, then it even increases the gear from four directly to six. It does take the maximum and minimum allowed speeds of the current road into account, ensuring that the speed of the vehicle remains within those boundaries.

The third and final strategy is the speed strategy. This strategy contains optimization which aims to limit the time spend driving from one location to another. If the controller tells the vehicle to increase its gear from four to five, then this strategy will only let that happen if it improves the speed of the vehicle, and only if the speed would remain below the maximum allowed speed of the current road.

4.3 RUM style Program

The RUM style program has moved the resource behavior of the propulsion, gearbox and vehicle out of their classes and into the PropulsionRUM, gearboxRUM and vehicleRUM classes. Instead of the strategy pattern, it has two optimizer classes and one OptimizerRUM.

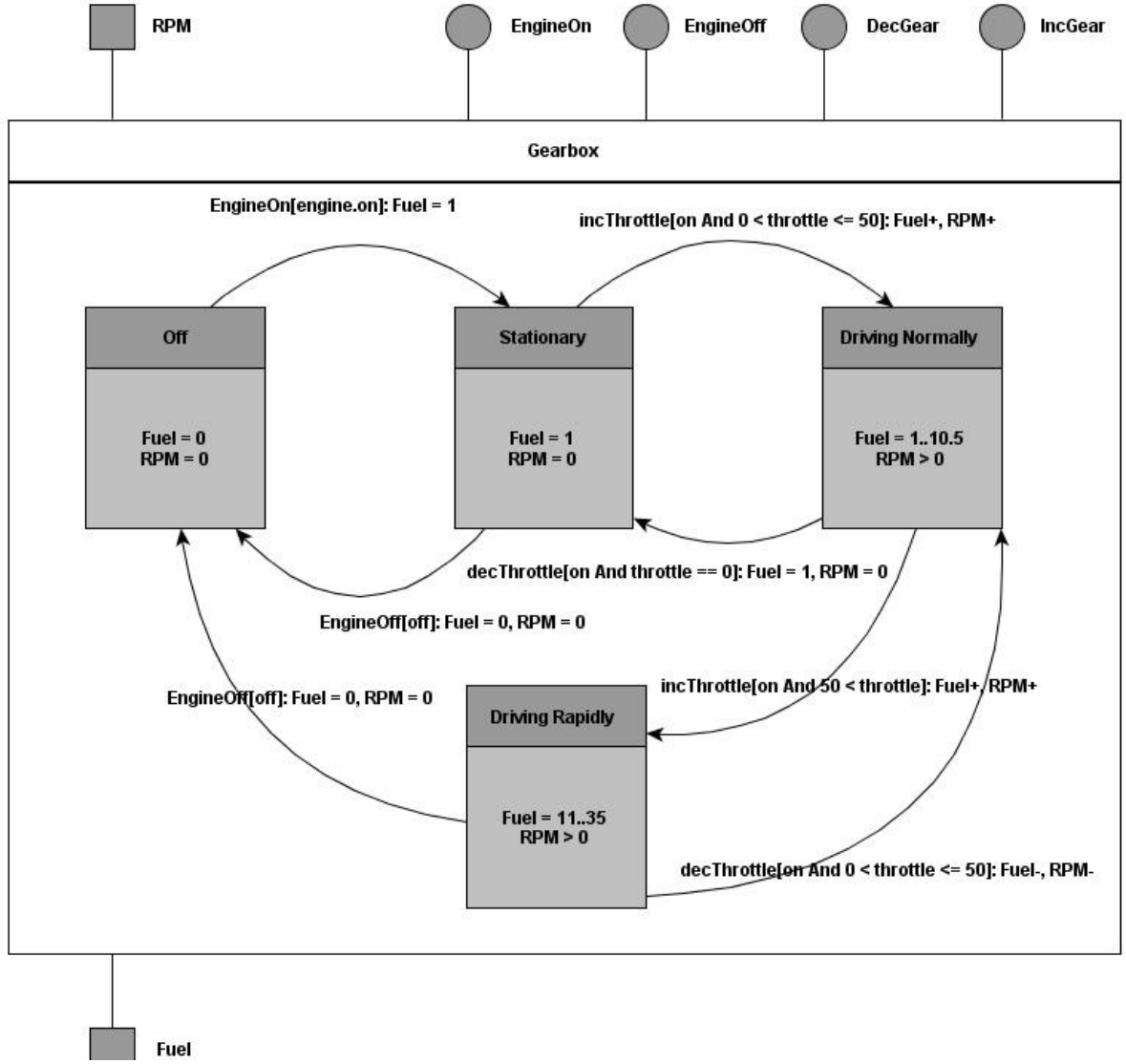


Figure 4.3.1 PropulsionRUM

The PropulsionRUM is responsible for managing the resource behavior of the propulsion. As shown in figure 4.3.1, it contains four different states, each with a different resource behavior. The resource which the RUM consumes is fuel, and the resource which it produces is RPM. The first state is the '*Off*' state, in this state the propulsion produces no RPM and consumes no fuel. The second state is the 'Stationary' state, in this state the propulsion still produces no RPM, but it does consume one litre of fuel per hour. The third state is the 'Driving Normally' state, this state represents that the vehicle is driving while using less than 50% of the maximum throttle. Both the produced RPM and consumed fuel depend on the throttle level. Lastly there is the 'Driving Rapidly' state, this state represents that the vehicle is driving while using more than 50% of the maximum throttle. The produced RPM is calculated in the exact same manner as in the 'drivingNormally' state, but the consumed fuel increases far more steeply.

This RUM intercepts four service invocations from the propulsion. These are the services responsible for turning the engine on and off, and the services responsible for increasing and decreasing the amount of throttle used by the propulsion. These services can cause transitions between the states as noted in figure 4.3.1. While figure 4.3.1 does not show it to keep the image clear, there is also a possible transition from the 'Driving Normally' state to the 'Off' state, similarly to the other transitions to the 'Off' state from the other states.

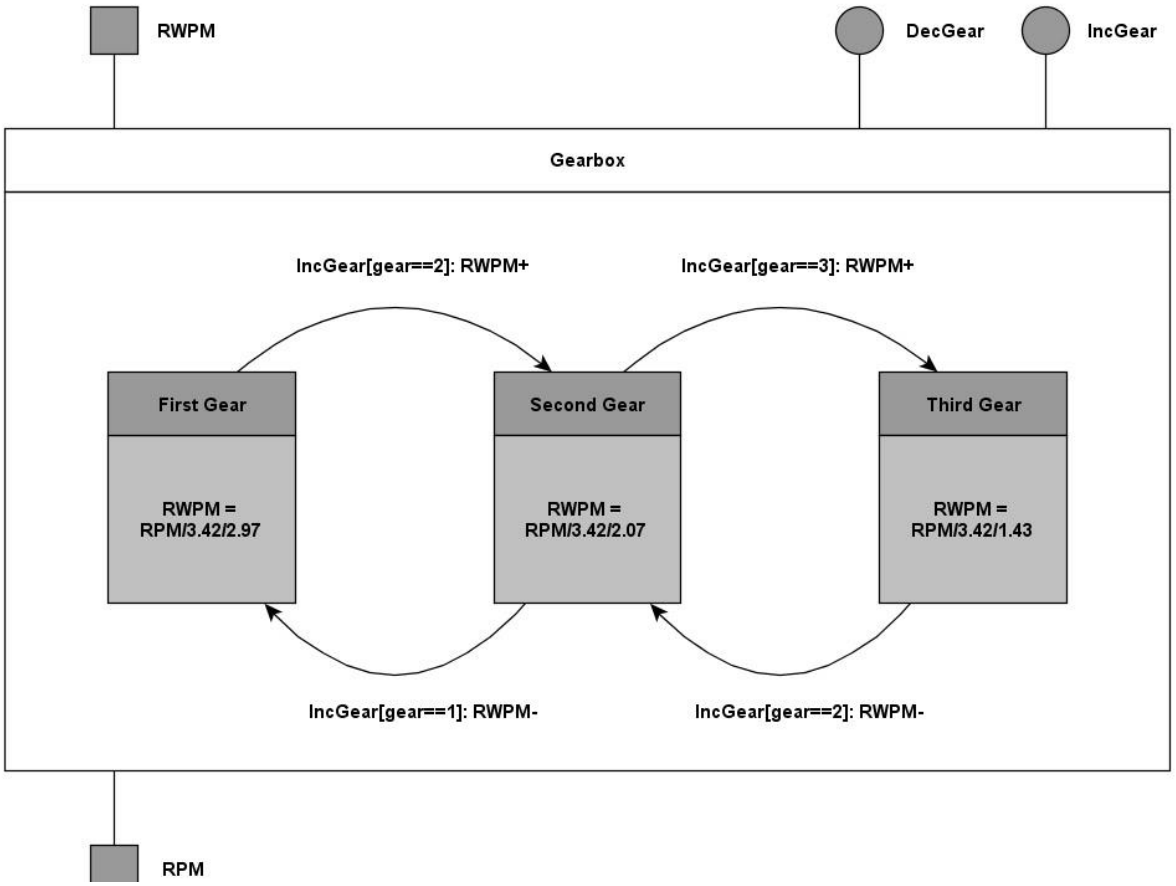


Figure 4.3.2 GearboxRUM

The GearboxRUM is responsible for managing the resource behavior of the gearbox. As shown in figure 4.3.2, it contains one state for each gear, each state having a different ratio. The actual gearboxes in the RUM and O-O programs have six gears, not three, but in order to keep the example clear we only show three in this figure. The resource which the gearbox consumes is RPM and the resource which it produces is rotations of the wheel per minute (RWPM). Each state has its own ratio, equal to the ratios of the gears in the Object Oriented program. The states all share the same differential ratio. In figure 4.3.2, this differential ratio is 3.42, while the first gear's own ratio is 2.97. This RUM intercepts two service invocations from the gearbox. These are the service responsible for increasing the gear, and the service responsible for decreasing the gear.

The VehicleRUM is responsible for managing the resource behavior of the vehicle. It only contains one state, as its resource behavior does not change. The resource which it consumes is RWPM, and the resource which it produces is speed, expressed in km/h. It does not have any state transitions since it only contains one state, due to this it also has no need to intercept the service invocations of a component.

Finally, two different optimizers exist, with similar goals to the strategy classes used in the Object Oriented program. One optimizer aims to keep the fuel consumption per driven kilometer at a minimum, while the other optimizer aims to maximize the produced speed. These optimizers also have a RUM. This RUM is responsible for interjecting the optimizer between the functional components. In order to optimize the resource behavior of the functional components, the optimizer's RUM interjects the optimizers between the vehicle and the gearbox, and between the vehicle and the propulsion.

5. Modularity Metrics

This chapter discusses which traits are commonly associated with modularity, and which modularity metrics which we have chosen in order to measure the modularity of the RUM and O-O programs.

Modularity aims to encapsulate related functionality into modules, decoupling it as much as possible from the rest of the program. By separating parts of the system into different modules, changes to one module are prevented from affecting other modules. These modules make the system more maintainable, easier to extend and less complex [7]. Since the RUM style aims to limit the impact on complexity, by modularizing the energy optimizers from the rest of the software, we have chosen to measure the complexity of the classes.

Two traits also commonly associated with complexity are cohesion and coupling. Cohesion measures how similar the methods in a module are. Cohesiveness of methods in a module is desirable, since it promotes encapsulation. Low cohesion also increases complexity, as this indicates that a module is responsible for more tasks than it should. This increases the likelihood of errors during development. Such modules should be split up into multiple different modules.

Excessive coupling is detrimental to modular design and makes it a lot harder to reuse classes. The larger the number of couplings, the higher the sensitivity to changes in other parts of the design, increasing the difficulty of maintenance. In order to improve modularity, and promote encapsulation, inter-object coupling should be kept to a minimum.

Next we discuss the metrics which we have chosen for measuring the complexity, cohesion and coupling of the RUM and O-O programs.

5.1 Complexity

We have chosen two metrics for measuring the complexity of a class, these are *Lines of Code* (LoC) and *Cyclomatic Complexity* (CC) [15] [13]. LoC is a simple count of the number of lines in a source program, or a class. We have chosen to exclude whitespace lines and comment lines, only including lines with actual code. LoC is very dependent on the used coding styles, which can be a disadvantage.

Cyclomatic Complexity is a widely used metric in software engineering for measuring the complexity of a program. Defined by Thomas McCabe, it is an easy to understand and calculate metric. The metric measures the number of linearly independent decision paths within a program. For example, a method with one if statement has two possible paths through the code, one which executes the statements within the if block and another path which doesn't.

There are several statements which increase the number of paths within a program. Some examples of such statements are: If, Else if, While and For. The original metric by McCabe treated every one of these statements as adding one new decision path. A variation of this metric also exists (extended cyclomatic complexity), which counts every Boolean operator as an extra decision path. For example, the if statement: 'if a and b', counts as two new decision paths in this version, while it counts as one path in the original metric. We have chosen to use extended cyclomatic complexity, as statements with multiple Boolean operators are more complex and error prone than statements without Boolean operators, and extended cyclomatic complexity thus gives us a better picture of the complexity of a method. Code blocks 5.1.1 and 5.1.2 show two different methods and their Cyclomatic Complexity.

Code block 5.1.1 contains a method which reduces the value of variable *z*, if variable *x* is true and variable *y* is greater than five. It also increases the value of variable *y* by three if variable *z* is still

greater than twelve after the first if clause. It then returns the multiplication of variables y and z. Line 2 contains an if statement and a Boolean statement, increasing Cyclomatic Complexity by two. Line 4 contains another if statement, further increasing Cyclomatic complexity by one. The total Cyclomatic complexity of method A is thus four.

```
1  def A(self):
2         if self.x and self.y >5:
3               self.z -= 9
4         if self.z > 12:
5               self.y += 3
6         return self.y*self.z
```

Code block 5.1.1

Code block 5.1.2 contains a method which increases the value of variable y as long as variable variable *i* is smaller than then and variable x remains true. If variable x is false then after this while loop variable z is returned, else variable y is returned. The while statement in line 3 also contains a Boolean statement, thus it increases Cyclomatic complexity by two. Line 7 contains an if statement, further increasing cyclomatic complexity by one. Line 9 contains an else statement, which doesn't increase the cyclomatic complexity as it doesn't add a new decision path, it is part of the if statement from line 7. Either line 8 is executed or line 10. Thus the Cyclomatic complexity of this method is also four.

```
1  def B(self):
2        i = 0
3         while i < 10 and self.x:
4               self.y += i
5               self.x = (self.y*self.z) < (8 * self.z)
6               i += 1
7         if not self.x:
8               return self.z
9         else:
10              return self.y
```

Code block 5.1.2

5.2 Cohesion

There are many different metrics out there for measuring cohesion. We have chosen three metrics, Lack of Cohesion in Method (LCOM), Tight Class Cohesion (TCC) and Loose Class Cohesion (LCC).

To help explain how these measures work, we have created figure 5.2.1. This example contains two classes: A and B. Methods are represented by rectangles, while instance variables are represented by circles. Class A contains three methods: *a, b* and *c*. Methods *a* and *b* use instance variable *x* and call method *c*. Method *c* does not use any instance variables. Methods *a* and *b* also call method *d* from class B. Class B also contains three methods: *d*, *e* and *f*. Method *d* uses instance variable *x*, method *e* uses instance variables *x* and *y* and method *f* only uses instance variable *y*.

LCOM is a metric by Chidamber and Kemerer [10], which measures the degree of similarity between methods in a class. It is computed as the number of method pairs which are dissimilar, minus the number of method pairs which are similar. A method pair is similar if both methods use at least one common instance variable. If the number of similar pairs is greater than the amount of dissimilar pairs then the value of LCOM is 0. A value of 0 thus represents a very cohesive class, while a higher value represents a less cohesive class, which can be split up into multiple smaller classes. The original version of LCOM has several problems. Often access to instance variables is restricted through

getters and setters, thus many methods end up using these getters and setters instead of accessing the instance variables directly. In LCOM, these methods are treated as dissimilar. There are also many cases where methods exist which do not directly access instance variables, but are coded entirely in terms of other methods of their class, which then do access these instance variables. In order to satisfy LCOM, these methods could certainly be merged into one big method, but all this achieves is to increase code duplication and the complexity of the methods in question. Figure 5.2.1 contains an example in class A, where improving cohesion would also increase code duplication. Method *c* uses no instance variables, but is called by both methods *a* and *b*. removing method *c* and moving its functionality into methods *a* and *b* would improve LCOM, while also adding undesired code duplication and complexity. So clearly this is not desired, and LCOM falls short when it is used to measure classes with such methods.

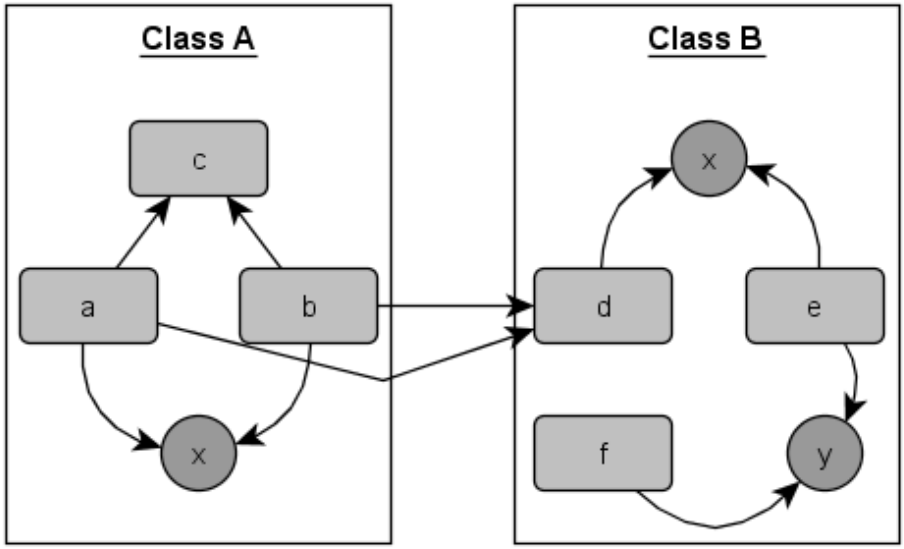


figure 5.2.1

A newer version of LCOM by Martin Hitz and Behzad Montazeri [12] addresses these issues. Their version of LCOM considers methods A and B to be cohesive if they access the same instance variable, or if A calls B, or if B calls A. After determining the related methods, a graph is drawn, connecting all related methods to each other. LCOM then equals the number of connected groups of methods in this graph. A value of 1 represents a cohesive class, where all methods are connected either directly or indirectly, while a value greater than 1 represents a class which can be split up into multiple smaller classes. Figure 5.2.2 shows the LCOM graphs for both classes from figure 5.2.1. Each graph contains one fully connected group, so both classes have an LCOM value of 1. While this version of LCOM shows into how many classes a measured class can be refactored, the value given by LCOM does not give any information about the size of the respective graphs. If two classes, C and D, both have an LCOM value of two and each have ten methods, with class C having two graphs with nine and one methods respectively, and class D having two graphs with five methods respectively, then clearly class C is much more cohesive than class D, but the metric does not show this. This is where our other chosen metrics, Tight Class Cohesion and Loose Class Cohesion, come in.

Tight Class Cohesion and Loose Class Cohesion [8] provide another way to measure the cohesion of a class, and are closely related to the idea behind Hitz and Montazeri's version of LCOM. TCC and LCC only consider methods which are visible, private methods are excluded unless they implement an interface or handle an event. Methods are considered connected if they both access the same instance variable, or if the call trees starting at methods A and B access the same instance variables

before leaving the class. Methods are considered indirectly connected if methods are not directly connected, but are connected through other methods, for example in figure 5.2.1, method *d* is connected to *e* and *e* is connected to *f*, thus *d* and *f* are indirectly connected through *e*.

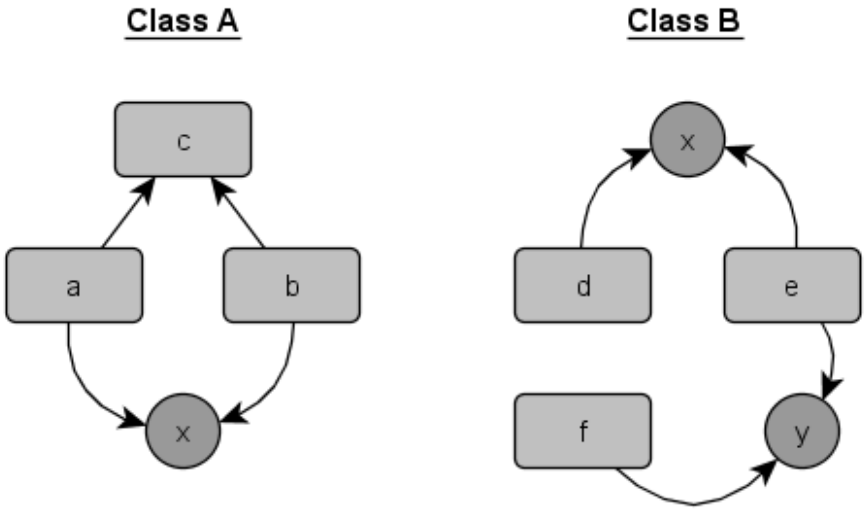


figure 5.2.2

The value of TCC is then computed by dividing the amount of connections by the maximum number of possible connections (NP), which is N*(N-1)/2, where N is the number of visible methods. The value of LCC is computed by adding the amount of direct connections to the amount of indirect connections, and then dividing this sum by the maximum number of possible connections. LCC is thus always greater than or equal to TCC. Figure 5.2.3 shows the connections between the methods for TCC and LCC, using the two classes from figure 5.2.1. Assuming that all methods are public, then both classes A and B have three public methods, so the maximum number of possible connections is three for each class.

Direct connections in figure 5.2.3 are represented by a straight line between two methods, with a circle interrupting the line, containing the name of an instance variable which both of the methods use. A direct connection can thus be seen between methods *a* and *b*, *d* and *e*, and methods *e* and *f*. Class A thus has one direct connection, class B has two direct connections. As can be seen, we have also added a class C, this class is similar to A, except that method c now calls methods *a* and *b*, instead of *a* and *b* calling *c*. The amount of direct connections in class C is three, as the call tree starting from method c includes both methods a and b, and thus includes access to instance variable *x*. Thus the TCC values for classes A, B and C are respectively: 1/3, 2/3 and 1. As for the indirect connections, class A does not contain any as the call tree from method *c* never comes in touch with any other methods. Class B does contain an indirect connection, methods *d* and *f* are indirectly connected through method *e*. Class C does not contain any indirect connections as all connections are direct. Thus the LCC values for classes A, B and C are respectively: 1/3, 1 and 1.

A class is considered to be non-cohesive when TCC<0.5 and LCC<0.5, while a class with an LCC of 0.8 is considered to be “quite cohesive” [8]. A class is maximally cohesive when TCC and LCC are both 1, as this means that all methods are directly connected. Class A is thus non-cohesive as both TCC and LCC are 1/3 for this class. But if method C were to be private, then it would be excluded from consideration when determining the TCC and LCC values. The amount of possible connections for class A would then drop to one, while the amount of direct connections would remain one. TCC and LCC would then both be one, which means the class would be maximally cohesive. Thus whether a

method is visible or not greatly affects the scores produced by TCC and LCC. Class B scored 2/3 for TCC and 1 for LCC, which means it's quite cohesive. Class C scored 1 on TCC and 1 on LCC, meaning it is maximally cohesive.

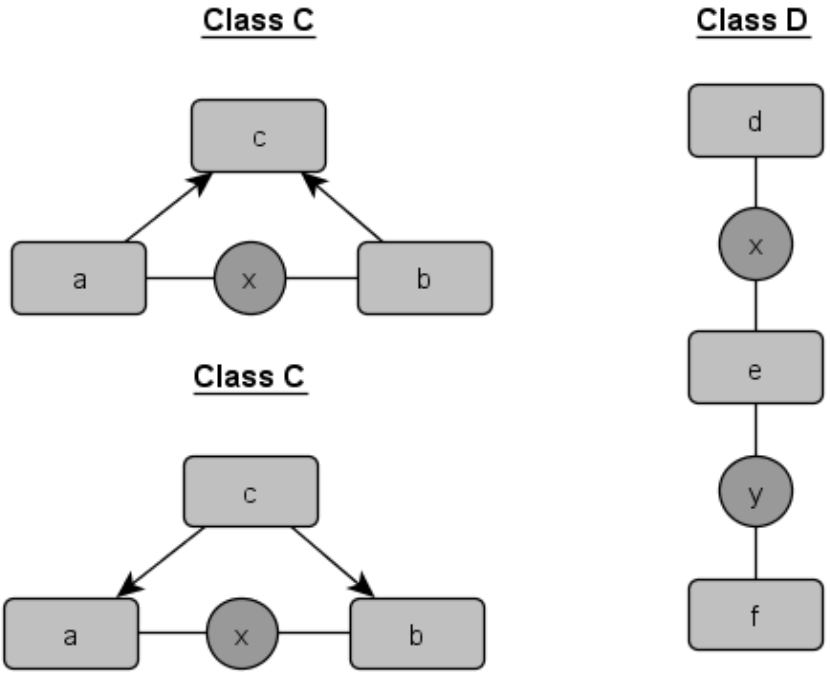


figure 5.2.3

As can be seen, one of the main differences between TCC/LCC and our chosen version of LCOM is seen when a class has a public method which does not access any instance variables, but is accessed by another method. LCOM considers these methods to be cohesive, but TCC and LCC do not. Both metrics thus have their advantages and disadvantages, LCOM shows how many different groups of connected methods exist in a class but it does not give information on the sizes of these groups, while TCC/LCC does not show exactly how many of such groups exist, but the results by these metrics can be useful to get a reasonable idea about the sizes of such groups. When combined, these metrics thus complement each other and give a much better picture of the cohesion of a class.

5.3 Coupling

We have chosen two metrics for measuring the coupling of a class, these are Coupling Between Objects (CBO) and Coupling Factor (CF).

CBO [10] is a straightforward metric for measuring coupling. Class A is coupled to class B when A uses methods or instance variables from class B. A high CBO is undesirable, as excessive coupling makes reuse harder. Sahraoui, Godin & Miceli [17] state that a CBO greater than 14 is too high. In figure 5.2.1, Class A has two methods which call a method from class B. In this example, class A thus has a CBO of one. Class B does not call methods from other classes and does not use instance variables of other classes, thus its CBO value is zero.

In Python, when a class wants to access its own instance variables or one of its own methods, it does so through the self-argument. In Python, it is a code convention to name the first argument of any non-static method 'self'. This argument always refers to the current instance. For example, to access the instance variable A, the following line of code would be used: 'self.A'. The usage of an instance variable or method from another class can thus be detected by looking for lines of code which access

instance variables or method, but do not have ‘self’ as the first preceding word. For example, ‘foo.A’, or ‘self.B.bar()’. In the first example the instance variable of an argument other than self is used, while in the second example a method of an instance variable is called. Both are cases of coupling, but since Python is a dynamically typed language, we cannot directly tell to what our current class is coupled as we do not know what B and foo are. It could very well be that B is an instance of the Foo class, while the argument foo is also an instance of the Foo class.

To compensate for these issues, all instance variables and arguments in our prototypes are properly named after the type of class that will be assigned to them. This allows us to determine the CBO value of each class.

The second coupling metric we used is CF. Coupling factor divides the amount of couplings of a class (the value generated by CBO) by the maximum possible amount of couplings. Thus, while CBO shows to how many classes another class is coupled, CF instead shows the amount of couplings in relation to the entire program. This can be useful to quickly spot outlier classes with high coupling. In large and very small programs CF becomes less useful, as the amount of possible couplings can become far too large or far too small. Take for example a program with 1001 classes, and thus 1000 possible couplings for every class. Class C is coupled to 10 classes, while class D is coupled to 15. Their respective CF values would be 0.01 and 0.015, both extremely low and very close to each other, but according to CBO the coupling of class D is far too high. Then take figure 5.2.1, if classes A and B are the only classes in the program, then A has a CF value of 1, while B has a CF value of 0. These values appear to be extreme, especially for class A, but it is actually coupled to only one other class. Thus the size of the program needs to be taken into account when determining what the value produced by CF implies.

6. Results

In this chapter we discuss the measurements we took, using the previously discussed metrics, of both the O-O and RUM style programs. We then compare the measurements of both programs to each other and discuss the results. In chapter four we discussed the structures of the two programs and showed what they have in common and where they differ. The parts of both programs which are completely the same have been excluded from the results displayed in this chapter, as all the measurement, besides CF, are entirely the same for those parts. This mainly involves the environment namespace of both programs, as the environment, with the exception of the World class, is not coupled to the rest of the program. Thus the results focus on the areas where the programs actually differ from each other.

6.1 Measurements

This section discusses the measurements taken of both programs. For each program, first the measurements of the complexity metrics will be discussed, then the cohesion metrics and finally the coupling metrics. Then the measurements of both programs are compared to each other and finally the RUM style library is compared to another language.

6.1.1 Object Oriented program

Table 6.1.1 shows the measurements of the two complexity metrics we used: the lines of code and the cyclomatic complexity, of the classes in the Object Oriented program.

Class	Lines of code	McCabe's CC
VehicleController	72	5
AbstractGearbox	21	1
Gearbox	13	1
AbstractGearboxStrategy	13	1
ManualGearboxStrategy	7	1
EfficiencyGearboxStrategy	33	8
PerformanceSpeedGearboxStrategy	36	6
AbstractPropulsion	36	1
Propulsion	30	3
AbstractPropulsionStrategy	13	1
ManualPropulsionStrategy	10	1
EfficiencyPropulsionStrategy	38	8
PerformanceSpeedPropulsionStrategy	39	6
AbstractVehicle	56	3
DeliveryVehicle	18	3
VehicleFactory	15	1

Table 6.1.1. Complexity measurements.

Figure 6.1.1 shows that the controller contains more lines of code than the other classes, and that it has an above average cyclomatic complexity in this program. The above average cyclomatic complexity is mainly caused by the methods which are responsible for moving a car from its current location to the location of a given package.

The functional components all show a low amount of lines of code and a cyclomatic complexity of three or lower. The cyclomatic complexity of the gearbox and propulsion is especially low, this is because these classes pass on most calls to their strategy instance.

Finally the strategy classes show a spike in cyclomatic complexity, while the amount of lines of code is average. The spike in cyclomatic complexity is caused by methods which are responsible for determining the optimal gear and throttle value for the vehicle.

Table 6.1.2 shows the measurements taken with the cohesion metrics. The abstract classes mostly contain abstract methods, which are implemented by their subclasses, therefor the cohesion of the abstract classes has not been included. The VehicleController scores excellently on cohesion, as each of its methods uses the controller's vehicle variable in some manner. The functional components, like the gearbox and propulsion, don't score nearly as well on cohesion. The Gearbox and Propulsion are less cohesive due to the strategy pattern. For example, the Propulsion class contains methods responsible for changing the throttle and for turning the engine on and off. These methods call methods from the strategy class, and the strategy class then determines how to change the throttle. The strategy class then also modifies the propulsion accordingly. But the Propulsion class also contains getter methods which calculate how much RPM the propulsion is generating, and how much fuel the propulsion is consuming. These getter methods simply use the variables of the propulsion class in a given formula, there is nothing to be optimized, and thus they were not included in the strategy classes. The same is the case in the Gearbox class. This causes the cohesion of the Gearbox and Propulsion to drop.

Class	LCOM	TCC	LCC
VehicleController	1	1	1
AbstractGearbox	-	-	-
Gearbox	2	0.333333333	0.333333
AbstractGearboxStrategy	1	1	1
ManualGearboxStrategy	1	1	1
EfficiencyGearboxStrategy	1	1	1
PerformanceSpeedGearboxStrategy	1	1	1
AbstractPropulsion	-	-	-
Propulsion	2	0.392857143	0.464286
AbstractPropulsionStrategy	1	1	1
ManualPropulsionStrategy	1	1	1
EfficiencyPropulsionStrategy	1	1	1
PerformanceSpeedPropulsionStrategy	1	1	1
AbstractVehicle	-	-	-
DeliveryVehicle	1	1	1
VehicleFactory	1	1	1

Table 6.1.2. Cohesion measurements.

The Strategy classes all score perfectly on cohesion. Every method in those classes accesses the same instance variable, usually the component which they are trying to optimize.

Finally table 6.1.3 shows the measurements taken with the coupling metrics. The VehicleController class has the highest amount of couplings, which comes as no surprise as it contains methods which are responsible for move the vehicle around, loading and unloading packages to the vehicle, refueling the vehicle, and determening which roads the vehicle shall take. It also increases the speed

of the vehicle when its current speed is below the maximum speed allowed on the current road, and it decreases the speed when the vehicle is going too fast. This ends up coupling the VehicleController to a lot of different classes.

Class	CBO	CF
VehicleController	10	0.285714286
AbstractGearbox	0	0
Gearbox	1	0.028571429
AbstractGearboxStrategy	4	0.114285714
ManualGearboxStrategy	1	0.028571429
EfficiencyGearboxStrategy	4	0.114285714
PerformanceSpeedGearboxStrategy	4	0.114285714
AbstractPropulsion	0	0
Propulsion	1	0.028571429
AbstractPropulsionStrategy	4	0.114285714
ManualPropulsionStrategy	1	0.028571429
EfficiencyPropulsionStrategy	4	0.114285714
PerformanceSpeedPropulsionStrategy	4	0.114285714
AbstractVehicle	5	0.171428571
DeliveryVehicle	5	0.114285714
VehicleFactory	2	0.057142857

Table 6.1.3. Coupling measurements.

Of the functional components, the gearbox and propulsion both have a very low coupling. They are each coupled to their strategy instances and that's it. The Vehicle classes have a higher coupling, as they are coupled to other functional components like the propulsion and gearbox, and they are also coupled to classes which model parts of the environment. The vehicle classes are coupled to these classes because they needs them in order to determine the current speed of the vehicle. As discussed in chapter four, the calculation of the current speed involves several factors, like the wind, the weight of the cargo, the steepness of the current road, the amount of throttle being used and the current gear of the gearbox. The vehicle needs to access the classes which contain this information, resulting in the above average coupling of both vehicle classes.

The strategy classes show a higher amount of coupling compared to most of the other classes. Taking the PerformanceSpeedGearboxStrategy class an example, this class uses several variables when determining the optimal gear for the current situation. It looks at the maximum allowed speed on the current road, the current speed of the vehicle, the amount of gears in the gearbox and also the current amount of throttle being used. This causes it to be coupled to four different classes. The DeliveryVehicle also has an above average amount of couplings in this program, this is mainly caused by the method which calculates the current speed of the vehicle.

6.1.2 RUM style program

Table 6.1.4 shows the lines of code and cyclomatic complexity measurements of the RUM style program. The VehicleController again has a high amount of lines of code and an above average cyclomatic complexity for this program. This is again mainly caused by the methods responsible for moving the vehicle from its current location to the location of a package. The functional components again show a low amount of lines of code and a low cyclomatic complexity. Their RUMs similarly have low amounts of lines of code and a minimal cyclomatic complexity.

A peak can be seen in the lines of code of the optimizer classes, their cyclomatic complexity is above average. The high lines of code values are mainly caused by the parsing of the results of the query system, as discussed in section 3.2.2. This parsing is taken care of by the abstractOptimizer, greatly increasing the amount of lines of code in this abstract class.

Class	Lines of code	McCabe's CC
VehicleController	73	5
AbstractGearbox	15	1
Gearbox	7	1
AbstractPropulsion	27	1
Propulsion	16	1
AbstractVehicle	58	2
DeliveryVehicle	12	3
VehicleFactory	20	1
AbstractOptimizer	127	5
FuelEfficiencyOptimizer	51	6
SpeedPerformanceOptimizer	41	5
GearboxRUM	18	1
AbstractGearboxGearState	11	1
GearboxFirstGearState	9	1
GearboxSecondGearState	13	1
GearboxThirdGearState	13	1
GearboxFourthGearState	13	1
GearboxFifthGearState	13	1
GearboxSixthGearState	9	1
PropulsionRUM	20	1
AbstractPropulsionState	14	-
PropulsionOffState	39	1
PropulsionStationaryState	24	1
PropulsionDrivingNormallyState	29	1
PropulsionDrivingRapidlyState	30	1
VehicleRUM	11	1
VehicleState	14	1
VehicleOptimizerRUM	9	1

Table 6.1.4. Complexity measurements.

Table 6.1.5 shows the cohesion metrics of the classes in the RUM style program. The VehicleController is again maximally cohesive. This time the functional components are also considered to be very cohesive. It is only the Propulsion component which does not score maximally on TCC, this is mainly caused by the methods responsible for increasing the throttle and the methods responsible for turning the engine on and off. These last two methods change the Boolean variable named *'on'*. The methods responsible for increasing and decreasing the throttle do not use this variable, they only modify the throttle value. LCC and LCOM are both still one because of the *stop* method. This method both turns the engine off and sets the throttle to the minimal value. Thus methods which use the *'on'* variable and methods which use the throttle variable are indirectly connected through the stop method.

The RUMs are all maximally cohesive, as are their state classes, with the exception of the PropulsionStationaryState. LCOM considers the PropulsionStationaryState class to be non-cohesive. TCC and LCC consider the PropulsionStationaryState class to be maximally cohesive. LCOM is 2 for this class because the private method, used to define the state transitions of this state, is not cohesive with the methods which are responsible for determining the resource consumption and production. This isn't the case in the other states because unlike those states, the PropulsionStationaryState uses constant values for its resource consumption and production. As shown in figure 4.3.1, while the vehicle is stationary it always consumes one litre of fuel per hour, while producing no RPM. The resource consumption and production of the driving states, depends on the value of the throttle variable from the propulsion class. In order to access the throttle variable, the resource related methods must access the propulsion instance variable. This variable is also used when defining the state transitions, thus LCOM is 1 for those classes, but 2 for the stationary state. TCC and LCC are still 1 because they do not include the method used to define state transitions, as this method is private and thus not visible. As shown in figure 4.3.1, while the propulsion is off its resource consumption and production are also constant, this has not caused a drop in cohesion for the PropulsionOffState because the AbstractPropulsionState also defines both resource related methods and returns default values of 0. The PropulsionOff state thus doesn't need to override these methods, while the PropulsionStationaryState has to override one of them.

Class	LCOM	TCC	LCC
VehicleController	1	1	1
AbstractGearbox	-	-	-
Gearbox	1	1	1
AbstractPropulsion	-	-	-
Propulsion	1	0.6	1
AbstractVehicle	-	-	-
DeliveryVehicle	1	1	1
VehicleFactory	1	1	1
AbstractOptimizer	1	1	1
FuelEfficiencyOptimizer	1	1	1
SpeedPerformanceOptimizer	1	1	1
GearboxRUM	1	1	1
AbstractGearboxGearState	1	1	1
GearboxFirstGearState	1	1	1
GearboxSecondGearState	1	1	1
GearboxThirdGearState	1	1	1
GearboxFourthGearState	1	1	1
GearboxFifthGearState	1	1	1
GearboxSixthGearState	1	1	1
PropulsionRUM	1	1	1
AbstractPropulsionState	-	-	-
PropulsionOffState	1	1	1
PropulsionStationaryState	2	1	1
PropulsionDrivingNormallyState	1	1	1
PropulsionDrivingRapidlyState	1	1	1
VehicleRUM	1	1	1

VehicleState	1	1	1
VehicleOptimizerRUM	1	1	1

Table 6.1.5. Cohesion measurements.

Class	CBO	CF
VehicleController	10	0.212765957
AbstractGearbox	0	0
Gearbox	0	0
AbstractPropulsion	0	0
Propulsion	0	0
AbstractVehicle	5	0.063829787
DeliveryVehicle	2	0.106382979
VehicleFactory	4	0.085106383
AbstractOptimizer	6	0.127659574
FuelEfficiencyOptimizer	7	0.14893617
SpeedPerformanceOptimizer	7	0.14893617
GearboxRUM	2	0.042553191
AbstractGearboxGearState	0	0
GearboxFirstGearState	0	0
GearboxSecondGearState	0	0
GearboxThirdGearState	0	0
GearboxFourthGearState	0	0
GearboxFifthGearState	0	0
GearboxSixthGearState	0	0
PropulsionRUM	2	0.042553191
AbstractPropulsionState	0	0
PropulsionOffState	1	0.021276596
PropulsionStationaryState	1	0.021276596
PropulsionDrivingNormallyState	1	0.021276596
PropulsionDrivingRapidlyState	1	0.021276596
VehicleRUM	1	0.021276596
VehicleState	6	0.127659574
VehicleOptimizerRUM	1	0.106382979

Table 6.1.6. Coupling measurements.

Finally there is Table 6.1.6, which contains the measurements of the coupling metrics. The VehicleController has an above average coupling just as before, as it still needs access to a lot of information in order to move the vehicle, load it with cargo and unload its cargo. The functional components show a minimal coupling, zero for both the Gearbox and the Propulsion, and only two for the DeliveryVehicle. Their RUMs have a slightly higher coupling, as they are both coupled to their component(s) and to their states. The states themselves vary in their coupling, the Propulsion states all have a coupling of one, as they use the throttle variable of their component in order to calculate their resource usage and consumption, while the Gearbox states have a coupling of zero because each gearbox state uses different constant values in order to calculate the resource consumption and production, and these values are stored inside the states instead of in their component. The

VehicleState shows a much higher coupling as many factors are involved in the calculation of the resource production, the vehicle's speed. It needs both the GearboxRUM and PropulsionRUM for this, and several of the environment related classes.

The optimizer classes also have an above average coupling as they need to access not just the vehicle, propulsion and gearbox components, but also their RUMs. The optimizers also need to access their optimizerRUM in order to gather information about the state machines of the other RUMs.

6.2 Discussion

Several differences can be seen between the measurements of the two programs. When it comes to Lines of Code the AbstractOptimizer of the RUM style has a lot more lines, compared to the strategy classes in the Object Oriented style. In section 3.2.2 we discussed the query capabilities offered by the Optimizer RUM, including the example query system. The optimizer in the RUM style uses the example query system implementation and needs to parse the results returned by this system, so the optimizer can determine which services it needs to invoke, and which arguments it should use when invoking these services. This parsing requires quite some lines of code.

Figure 6.1.2.1 shows that the cyclomatic complexity of the AbstractOptimizer is only five, which is in line with its subclasses and slightly lower than the Efficiency and Speed Strategy classes from the Object Oriented style, as shown in figure 6.1.1.1. The cyclomatic complexity shows that the methods in the optimizers are slightly less complex than those in the strategy classes, though the difference negligible, differing only by two points. Improvements to the example query system, so that the optimizer receives clearer and more direct results could greatly help cut down on the Lines of Code in the AbstractOptimizer.

As can be seen when comparing figures 6.1.1.1 and figures 6.1.2.1, the functional components, like the gearbox, propulsion and vehicle, have less lines of code in the RUM style program. The propulsion also has a lower cyclomatic complexity. This is because their resource behavior related functionality was moved to the RUMs and their states, but these new RUMs and their states all have a minimal cyclomatic complexity of one and a low amount of lines of code. In the Object Oriented program the resource related getter methods contained several if clauses, as the returned value differed depending on whether the engine was on or off, etc. In the RUM style program these if clauses are instead represented as the different states of the RUM. Thus the overall complexity of the functional components and their resource behavior has been reduced. In general this should also occur in other programs where the resource behavior differs depending on certain variables in the component, like whether the engine is turned on or not. If a state still contains an if clause in one of its resource related methods, then this indicates that the state can be split into multiple other states.

As can be seen in figures 6.1.1.2 and figures 6.1.2.2, when it comes to the Cohesion measurements, the Gearbox and Propulsion both show improvements in the RUM style program. Both were non-cohesive in the Object Oriented program, but are very cohesive in the RUM style. The other classes are maximally cohesive in both styles. This difference is mainly caused by the strategy pattern. In the Object Oriented program, all the functional concerns of the components are handled by the strategy classes, while the components themselves still contain several getter methods which do not interact with the strategy instance variable of the component.

Finally, when it comes to coupling, the functional components like the gearbox, propulsion and vehicle have a much lower coupling. With the gearbox and propulsion the coupling was originally caused by the strategy pattern. In the RUM style these components are no longer coupled to

components with optimization functionality. The coupling has been reduced in the vehicle because it was originally coupled to several classes so it could calculate its own speed, which is a resource matter. The VehicleState now calculates the speed produced by the vehicle, and because the resource behavior of the propulsion and throttle components are now located in their RUMs instead of in those components themselves, the coupling of the VehicleState is higher than that of the DeliveryVehicle in the Object Oriented Style.

The Object Oriented program has an advantage when it comes to creating its optimizers, as they are coupled to fewer components. For each functional component to which the optimizer is coupled in the RUM program, it is also coupled to at least one RUM containing the resource behavior of the component. On the other hand, the RUM style sees a reduction in the coupling of the functional components, as they are no longer coupled to their optimizers. The RUM style has the advantage that the functional side of the program can be written without considering the optimizers, as thanks to the Interjector module optimizers can be interjected between any two functional components, while in the Object Oriented program a functional component needs to be modified in order to incorporate an optimizer. These advantages and disadvantages also apply in general, but the increased coupling of the optimizers only applies for components which the optimizer needs to optimize. If an optimizer is also coupled to several other components, but only needs to know about their resource behavior and doesn't need to modify this behavior, then the optimizer would only need to be coupled to this component's RUM and not to the component itself, while in the Object Oriented version it would only be coupled to the component. The functional components won't need to be coupled to optimizer components in general either, as instead of the functional component calling the optimizer, the optimizer's RUM takes care of redirecting all relevant calls. Thus in general, the functional components which need to be optimized will have a lower coupling, while the optimizer components will have a slightly higher coupling, and can be interjected seamlessly between any two components.

6.2.1 Other languages

During our previous investigation into suitable programming mechanisms for the RUM Library [18] we investigated other languages besides Python, one of these was Java. We will shortly discuss where this language could do better than our current Python Library, and where it would be less beneficial.

When we investigated Java we found aspects to be suitable for intercepting service invocations. Java has several Aspect Oriented Programmaing (AoP) language extensions available for it. We investigated AspectJ [14] as it is one of the more well-known AoP language extensions for Java. In AspectJ, aspects are used to add functionality before and after existing methods of your choice. These aspects contain pointcuts [5], which are used to define which methods to intercept. An example of such a pointcut, which is used to intercept the incThrottle and decThrottle services, can be seen in code block 6.2.1. Advices [2] are used to define what functionality to add before/after an intercepted method. There are three types of advices: the *before* advice, which only adds functionality before an intercepted method is executed, the after advice, which only adds functionality after an intercepted method has been executed, and the around advice, which can be used to add functionality before and after the execution of the intercepted method. The around advice can also be used to prevent the execution of the intercepted method, as the intercepted method must be executed manually inside an around advice. An example of this can be seen in line 9 of code block 6.2.1.

Code block 6.2.1 shows an aspect which contains one pointcut and one advice. The pointcut at line three intercepts the service invocations of the Propulsion's engineOn and engineOff services. The

around advice at line seven uses this pointcut. The call to *component(p)* on line seven is a method which checks if the propulsion instance in question is the instance in which this aspect is interested. If this is the case then the method invocation is intercepted and the around advice is executed, else the around advice does not execute. It is thus used to filter out all other Propulsion instances. At line nine the intercepted service is invoked.

The current Python RUM library's method wrappers also add code before and after the wrapped method and are thus functionally similar. Inside the around advice it is also possible to find out from

```
1  public aspect PropulsionRUMAspect extends RUMAspect<Propulsion>
2  {
3        pointcut throttle(): target(Propulsion) &&
4                    (call(void engineOn()) ||
5                          call(void engineOff()));
6
7        around(Propulsion p): component(p) && throttle() {
8              \\add code to be executed before the intercepted method here
9              proceed(amount)
10             \\add code to be executed after the intercepted method here
11       }
12 }
```

Code block 6.2.1

which instance the intercepted call originated, and to then redirect the intercepted call to an optimizer. This is similar to what the Interjector's wrapper does with the stack frame objects. Thus AspectJ's aspects support all the functionality required for the interception of service invocations and for the interjection of optimizers.

AspectJ also has functionality which supports one of the RUM library's current limitations mentioned in 3.1.1, namely the precedence system. AspectJ already contains a precedence system for aspects. Code block 6.2.2 show an example of such a precedence declaration. When it comes to the precedence between around and before advices, the aspect with the highest precedence is executed first. When it comes to the precedence between around and after advices, the aspect with the lowest precedence is executed first. When it comes to precedence purely between around advices, the aspect with the highest precedence wraps around the aspect with the second highest precedence, and so on. Similarly to how method wrappers can wrap other method wrappers.

```
1  public aspect AspectOrdering
2  {
3      declare precedence : around1, around2, before3, after4;
4  }
```

Code block 6.2.2

The disadvantage to this precedence system is that it is aspect based, thus if an aspect contains several advices then they will all follow the given precedence declaration. If one advice is supposed to be first at all times, but another is supposed to always be last, then the aspect will need to be split up into two different aspects. One aspect can then no longer contain the advices for one RUM, multiple aspects would instead be required for this, resulting in scattering. With the Python RUM Library there is room to implement a precedence system on a service level instead of just on a RUM level. If a RUM intercepts two or more services, then it could be possible for this RUM to declare a different desired precedence for each of these services.

AspectJ also contains a wildcard system, which can be used when defining pointcuts. As an example, take code block 6.2.3, and compare this pointcut to the one specified in code block 6.2.1.

```
1  aspect PropulsionInterceptor {
2        pointcut throttle(): target(Propulsion) &&
3                    (call(void *(int)));
4  }
```

Code block 6.2.3

The above pointcut intercepts all methods with the return type of void and one argument of type integer, belonging to the Propulsion class. This enables the ability to intercept multiple service invocations with one pointcut, or to interject an optimizer between multiple methods with one pointcut. If the *component(p)* call from code block 6.2.1, line 7, is removed then the advice can also be deployed on a class level instead of an instance level. This enables us to interject an optimizer on a class level instead of on an instance level. The optimizer would automatically be interjected whenever a new instance of the given class is created. Our Python Library currently can only interject an optimizer at an instance based level, one instance at a time.

One still unmentioned advantage which Python has advantage over AspectJ is that Python is a dynamically typed language, where everything is also an object. This makes Python very expressive and quite powerful, enabling us to reuse the methods in the Library when implementing RUMs, and greatly cutting down on the needed lines of code in the RUMs. An example of this can be seen in code block 6.1.4. The first part of this code block contains two pointcuts, one which intercepts several service invocations with the return type of void, and a second pointcut which intercepts a service invocation with the return type of Boolean. Differentiating between these return types is important for around advices, if an around advice for example wraps around a method which returns a Boolean value, then the around advice needs to pass on a Boolean value as well, as the method which originally called the intercepted method is expecting a value of the type Boolean to be returned to it. Lines 11 and 17 also show that the around advices declare their return type, void in

```
1  public privileged aspect PropulsionRUMAspect extends RUMAspect<Propulsion>{
2        private pointcut engine():
3              call(void *.engineOn()) ||
4              call(void *.engineOff()) ||
5              call(void *.decThrottle(*))||
6              call(void *.incThrottle(*));
7
8        private pointcut engineIsOn():
9              call(boolean *.isOn());
10
11       void around(Object e) : component(e) && engine(){
12             standardBefore(e);
13             proceed(e);
14             standardAfter(e, thisJoinPoint.getSignature());
15       }
16
17       boolean around(Object e): component(e) && engineIsOn(){
18             standardBefore(e);
19             boolean result = proceed(e);
20             standardAfter(e, thisJoinPoint.getSignature());
21             return result;
22       }
23 }
```

Code block 6.2.4

the case of the first advice and Boolean in the case of the second. Lines 14 and 20 use the *thisJoinPoint.getSignature()* method in order to retrieve the name of the intercepted service, and then pass it on to the RUM through the *StandardAfter* method.

As can be seen in code block 6.2.4, with AspectJ both a pointcut and advice need to be defined for every RUM. It is not possible to reuse one advice for every RUM, because every RUM uses a different pointcut in its advice and because some of the intercepted methods have a return type other than void. The more different return types there are between the service invocations which need to be

```
1  self.registerToServiceInvocation(propulsion, propulsion.incThrottle)
2  self.registerToServiceInvocation(propulsion, propulsion.decThrottle)
3  self.registerToServiceInvocation(propulsion, propulsion.engineOn)
4  self.registerToServiceInvocation(propulsion, propulsion.engineOff)
5  self.registerToServiceInvocation(propulsion, propulsion.isOn)
```

Code block 6.2.5

intercepted, the more pointcuts and advices will need to be defined. With our current Python Library only one method needs to be used in order to intercept a service invocation. For the user, everything is contained inside the library and the *registerToServiceInvocation* method can be used in order to intercept service invocations. The return types of these methods do not matter in any way.

The same is the case for interjecting optimizers. With the Python library one method call, to one of the library's methods, is sufficient to interject an optimizer between two instances, but with AspectJ another pointcut and advice need to be created, where the pointcut defines all the methods for which calls need to be redirected to the optimizer, and where the advice needs to access the logic which determines if the call should be redirected or not. The advice is then even responsible for redirecting the call to the optimizer when necessary, while in the RUM Library this all happens under the hood inside the library itself. Python's dynamic typed system, together with everything being an object, is thus an advantage it holds over AspectJ.

7. Possible Improvements

This chapter discusses several possible improvements for the RUM Library, some of these were already shortly discussed previously, like the precedence system in chapter three.

Class level optimizer interjection. As discussed in chapter three, the interjection of an optimizer currently happens between two instances. An optimizer can be interjected between instance A and B, so that all calls from A to B are redirected to the optimizer. It is also possible to specify that only calls to specific methods of instance B should be redirected. If there are several instances of the same functional component, and all of these need to be optimized by the same optimizer, then we currently need to interject the optimizer once for each instance. If a new instance of this component is instantiated later on, then we need to interject the optimizer again, so it can also optimize this new instance. The ability to interject optimizers on a class level can automate this process. An extension to the Interjector module could enable the RUM to automatically wrap new instances of a class at the moment that they are instantiated. This can be achieved by wrapping the __init__() method (the constructor) of the target class in a method wrapper. This method wrapper will then automatically be called when the __init__() method is invoked during the instantiation of the new instance. The wrapper can then invoke this __init__() method as normal, and afterwards it can warp the desired methods of this new instance. Of course the __init__() method is optional, a class only needs one if it needs to set certain variable's values, or execute certain methods, at instantiation. If a class does not have an __init__() method defined then there's nothing to wrap. In this case an __init__() method can be added to the class in the same manner as how method wrappers are added, using Python's built-in setattr() function.

Automatic re-deployment of wrappers when a wrapper is replaced by a new, unwrapped, method object. Another possible improvement, which also involves the interjection of optimizers, can be seen in the following scenario. When an optimizer has been interjected between instances A and B, in order to redirect all calls from A to B to an optimizer, and one or more of the methods from instance B are changed, replacing an old method object with a new method object, then this new method object will most likely not have been wrapped. A call from instance A to this method in B will then not be redirected to the optimizer. If it is desired to automatically wrap such methods, so calls from instance A to these methods are also redirected to the optimizer, then further improvements to the Interjector module are required. Any assignment statement, such as 'self.x = 5', and an assignment statement where an old method object is replaced by a new method object, are all internally handled by Python's built-in setattr() function. So, when an optimizer is interjected between instances A and B, then the setattr function of instance B could also be wrapped. This wrapper can then be used by the Interjector to detect when the above scenario occurs, and to then automatically wrap the new unwrapped method. Of course it is also possible that the Interjector's wrapper was simply wrapped by the Interceptor, but this can be detected. In this case nothing needs to be done. This improvement is also applicable to the Interceptor module, as the same scenario could occur there for an intercepted service.

Detecting static caller methods when redirecting calls to optimizers. Another improvement which could be investigated also involves the Interjector. Take the following scenario. An interceptor has been interjected between components A and B. A third component, C, exists. This component's class has a static method, whose first argument is component A. This static method calls a method belonging to component B. When this static method is evoked, the Interjector's wrapper will be called when the method belonging to component B is called. When the Interjector examines from which component the call originated it will examine the first argument of the calling method, as this is the 'self' argument for non-static methods. It will then determine that the call originated from

component A, while it didn't originate from there at all. This error occurs because static methods do not have a 'self' argument and the Interjector is currently not capable of discerning whether the calling method is a static method or not, as the Interjector cannot access the actual method object of the calling method through the stack frame object, it can only access the code object containing the compiled function bytecode of this object. An investigation could be done into possible solutions for detecting whether a calling method is static or not.

Replacement for stack from dependency. Another improvement which could be investigated is whether it is possible to determine where a method call originated from, without requiring Python's stack frames. As mentioned in section 3.2.1, not all Python interpreters are guaranteed to support Python stack frames, the interjector module won't function correctly if the used interpreter doesn't support stack frames. CPython [3] is an example of such a Python Interpreter, as it does not currently support stack frames. Thus in order to increase the compatibility of the library, an alternative to Python's stack frames could be investigated.

Precedence system for service invocation interception. Finally, as mentioned in section 3.1.1, a precedence system could be added to the Interceptor module, allowing a RUM to define a precedence either on a RUM level, similarly to the precedence system in AspectJ, or on a service level, so a RUM can define different precedence desires for each of the services which it intercepts. If a specific RUM needs to check for, and execute, state transitions before another RUM, then this can in the currently library only be done by having the first RUM declare the service invocations which it wishes to intercept, before the other RUM declares which it wants to intercept. But as all the information about which RUM is interested in which service invocations is currently stored in one place, it is possible to extend the Interceptor by also storing precedence related information and using it to determine in which order to inform the RUMs about a service invocation interception.

8. Conclusion

Our goal was to investigate the modularity of a program implemented in the RUM style. For this we specified four requirements of a development environment in chapter two. We have implemented this development environment as a library in Python. We have implemented a program in the object oriented style, and another functionally equivalent program using the created library, and measured the modularity of both. Below, we discuss to what extend the implemented development environment meets the requirements and whether Python offered sufficient functionality and expressiveness for the development of this library. Finally we discuss the modularity of programs developed in the RUM style.

A RUM needs to be capable of intercepting service invocations. As discussed in section 3.1.1, the Interceptor module allows for the interception of any desired service invocation through the usage of method wrappers. The Interceptor creates a method wrapper which wraps the desired service, and replaces the reference to this service with a reference to the wrapper. Any time the service is invoked, the wrapper is invoked instead, allowing the wrapper to both invoke the wrapped service and to inform any RUMs about the service which was invoked. The implemented Interceptor module thus allows a RUM to intercept any service invocation it desires.

It needs to be possible to declare a RUM's state transitions in such a way, that they are introspectable. As discussed in section 3.1.2, state transitions are defined by a guard and a next state. This guard has the structure of an AST tree, which can contain operands, operators and other guards. Classes have been created for the operands, operators and guards, containing variables which contain important information about the entity. This ensures that all the information which is supplied to such an object upon instantiation is stored inside it, for later usage and introspection.

The library needs to allow for the interjection of optimizers between functional components. As discussed in section 3.2.1, the Interjection module allows for the interjection of an optimizer between any two components, or between a component and a method of choice from another component. The Interjector creates and deploys a method wrapper, which wraps the method(s) of the component which needs to be optimized. Any time one of these wrapped methods is called, the Interjector will check if the call needs to be redirected to the optimizer, and redirects it if necessary. The Implemented Interjector module thus allows for the interjection of optimizers between any components, including functional components.

The library needs to allow for an optimizer to introspect a RUM's state transitions. As discussed in section 3.2.2, an example query system has been implemented, which shows how a RUM's state transitions can be introspected in order to determine which actions are necessary in order to cause a specific state transition. Thus our implementation of this library allows an optimizer to introspect a RUM's state transitions.

From there requirements we can conclude that the implemented Python library contains the functionality required for the RUM style, and that a dynamic language like Python offers enough functionality and expressiveness for an implementation of the RUM style without requiring other 3rd party libraries or Aspect Oriented Programming language extensions.

The investigation into the modularity of the RUM style in Python has shown that the RUM style is quite modular. The functional components showed an improvement in coupling as they are no longer coupled to optimizers, while the optimizers show a slight increase in coupling as they are no longer just coupled to a functional component which they optimize, they are now also coupled to the RUM of that component. In return this does allow for the seamless interjection of an optimizer

between any two components at runtime, and these optimizers can also seamlessly be removed at any time. The complexity measurements showed no difference between the RUM style program and the Object Oriented program, and the cohesion measures showed only a slight improvement for the functional components which had strategy instances in the Object Oriented program.

Thus, based on the results of the investigation, a program in the RUM style in Python is at least as modular as a comparable Object Oriented program. At the same time the RUM style allows for the interjection of optimizers, at runtime, between any two components, without those components becoming coupled to the optimizers.

Bibliography

[1] Fletcher, M.C. Metaclasses, Who, Why, When (2004). Retrieved May 20, 2014: http://www.webcitation.org/5lubkaJRc.

[2] Advice. Retrieved November 13, 2014: https://www.eclipse.org/aspectj/doc/next/progguide/semantics-advice.html.

[3] Cython: C-Extensions for Python. Retrieved November 13, 2014: http://cython.org/.

[4] Data Model. Retrieved November 4, 2014: https://docs.python.org/3.3/reference/datamodel.html.

[5] Join Points and Pointcuts. Retrieved November 13, 2014: https://eclipse.org/aspectj/doc/next/progguide/language-joinPoints.html.

[6] Welcome to Python. Retrieved October 20, 2014: https://www.python.org/.

[7] Albin, S.T. *The Art of Software Architecture: Design Methods and Techniques*. John Wiley & Sons, 2003.

[8] Bieman, J. M., Kang, B. Cohesion and reuse in an object-oriented system. *Proceedings of the 1995 Symposium on Software* (1995), 259-262.

[9] Brinke, S., Malakuti, S., Bockisch, C., Bergmans, L., Aksit, M. A Design Method For Energy-Aware Software (November 2012).

[10] Chidamber, S. R., Kemerer C.F. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20, 6 (June 1994), 476-493.

[11] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, 1995.

[12] Hitz, M., Montazeri B. Measuring Coupling and Cohesion in Object-Oriented Systems. *Proceedings of International Symposium on Applied Corporate Computing* (1995), 25-27.

[13] Jay, G., Hale, J.E., Hale, D.P., Kraft, N.A., And Ward, C. Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship. *JSEA*, 2 (2009), 137-143.

[14] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G. An Overview of AspectJ. In *ECOOP '01 Proceedings of the 15th European Conference on Object-Oriented Programming* (Londen 2001), Springer-Verlag, 327-353.

[15] McCabe, T. A complexity measure. *IEEE Trans. On Software Engineering*, 2, 4 (December 1976).

[16] Royce, W. Improving software economics-top 10 principles of achieving agility at scale. *White Paper, IBM Rational.* (May 2009).

[17] Sahraoui, H.A., Godin, R., Miceli, T. Can Metrics Help Bridging the Gap Between the Improvement of OO Design Quality and it's Automation? *ICSM 2000 Proceedings of the International Conference on Software Maintenance* (2000), 154-162.

[18] Windhouwer, D. Overview of Programming Language Support for RUMs (June 2014).

A. RUM Library

The code for the RUM Library, the RUM vehicle program and the Object Oriented vehicle program is available at: https://github.com/S0166251-FinalProject/RUM