Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2019) 27: 3615 – 3632

doi:10.3906/elk-1812-164

Reusable and interactive classes: a new way of object composition

Saeid MASOUMI, Ali MAHJUR*
[1]Department Computer Engineering, Faculty of Electrical and Computer Engineering,
Malek-Ashtar University of Technology, Tehran, Iran

Received: 23.12.2018 • **Accepted/Published Online:** 27.05.2019 • **Final Version:** 18.09.2019

Abstract: Separating object features from base classes is one of the popular ways of software development. Some popular programming approaches like object-oriented programming, feature-oriented programming, and aspect-oriented programming follow this approach. There are four advantages of using features: 1) Instantiability: the ability to create instances of a feature, 2) Reusability: the quality of a feature being reusable in many compositions, 3) Loosely coupled composability: the ability to compose/decompose features easily at object instantiation time (not offering new data types for compositions), and 4) Interactability: the ability of a feature to crosscut (interact with) other features inside the object. Existing approaches do not find strong evidence to support these advantages altogether. In this paper, we propose a new approach that provides all the advantages mentioned above. In our approach, each feature is developed as a class that can be instantiated or reused. A new composition method is also proposed to compose features of an object where it is instantiated. In such a way, a feature can be either a complete object or part of a big object. In fact, composing different reusable features yields object variations, since features can be easily added/removed in a loosely coupled manner. To make features interactive, we augment them with events. Events provide the interactions among the different features of an object. We show that events are soft dependencies that do not affect the reusability of features while method callings in inheritance-based models do.

Key words: Instantiability, reusability, loosely coupled features, feature composition, event-based interaction, crosscutting feature

1. Introduction

Class is the basis of the object-oriented programming (OOP) paradigm. Usually, a class encapsulates a number of roles. Each role, called a feature, represents one of the functionalities of the class. This way, all objects instantiated from a class have all the features of the class.

Basically, novice programmers prefer to make a single big class that includes any needed and future features. It was previously thought that such a full-featured class, as a generator of instances, is sufficient for any use. However, time showed that it is not the most appropriate element to reuse. In fact, a class, as a unit of reuse, should be small. Therefore, reusability and full-featureness often conflict.

The suitable solution is to separate the development of features from classes. Inheritance-based approaches (like single/multiple inheritance, mixin [4–6]), Role-based approaches (like mixin-layers [22, 23], ObjectTeams/Java [8], and J& [16]), approaches based on separation of concern (like aspect-oriented programming (AOP) [11], Hyper/J [24], and FOP [18–21]), and step-wise refining models (like refinement [2]), have had the

*Correspondence: mahjur@mut.ac.ir

same idea. They allow programmers to have features apart from classes and compose them on demand.

However, there is some criticism directed at them. We show some of the main shortcomings through an example in which Queue and Stack are considered as base classes and Counter and Lock as features (Figure 1).

- **Queue** (Figure 1a) provides enqu and dequ operations on a queue,
- **Stack** (Figure 1b) provides push and pop operations on a stack,
- **Counter** (Figure 1c) adds a local counter (used for the size of a list), and
- **Lock** (Figure 1d) adds a switch to allow/disallow modifications of an object

```
class Queue{
   void enqu(int value)
   {...}
   int dequ()
   {...}
}
```
(a)

```
class Stack{
   void push(int value)
   {...}
   int pop()
   {...}
}
```
(b)

```
class Counter{
  int cnt=0;
  void inc(){cnt++;}
  void dec(){cnt--;}
  int size(){return cnt;}
}
```
(c)

```
class Lock{
  bool l=true;
  void lock(){l=false;}
  void unlock(){l=true;}
  bool is_unlocked(){return l;}
}
```
(d)

Figure 1. Queue and Stack are base classes and Counter and Lock are features.

The first shortcoming is feature dependency. In recent works, like AOP or refinements, a feature is defined as a dependent and incomplete entity. In this way, it has meaning just in the presence of a base class. This makes it unsuitable for instantiation. For example, when Counter is developed as a feature (such as an aspect or a refinement) of Queue, to get a pure Counter, the programmer is not allowed to instantiate this feature. Instead, he has to create another Counter as base class (not feature), causing code duplication. The same thing happens to other features like Lock, Log, and Security.

The second shortcoming is feature reusability. In single inheritance, AOP, and refinements, since a feature is attached to a specific base class, it is impossible to reuse a feature for other base classes of the same family. Again, Counter and Lock as features of Queue cannot be reused for Stack, Tree, etc.

The last shortcoming is that current approaches do not support crosscutting features. Features usually crosscut each other by injecting some codes into some specified points. Primitive feature composition approaches, like aggregation, do not let features to crosscut each other at all. Usually, there is a composing feature which accumulates other features and puts them into a new data type and then handles their interactions (Figure 2).

AOP is one of the most famous approaches in this context. Although aspects as features in AOP crosscut classes, they are neither instantiable nor reusable. Moreover, an aspect cannot crosscut other aspects. It just crosscuts base classes.

Other approaches (like single inheritance, mixin, mixin-layers, and J&) use inheritance to handle feature interactions. In this way, a feature inherits another feature to crosscut its methods. Of course, there is always a hard dependency between child and parent features in the inheritance hierarchy since child features hard-code

```
class LockCounterStack{
  Stack s=new Stack();
  Counter c=new Counter();
  Lock l=new Lock();
  void push(int value){
    if (l.is_unlocked()){
      s.push(value);
      c.inc();
    }
  }
}
```

Figure 2. Lack of crosscutting features in aggregation.

the callings of parent methods. This notably lowers the reusability of a child feature since it is tightly coupled to its parents.

As a result, recent works are insufficient to provide instantiable, reusable, and crosscutting features altogether. To enrich features, this paper considers a feature as an independent class, which can be instantiated alone. Moreover, we introduce a new way of feature composition. Our composition mechanism provides a variation of objects and improves reusability, since our composable features can participate arbitrarily in many compositions. Moreover, to support crosscutting features, instead of method calling or overriding, this paper proposes event raising which signals the states of feature methods to other features in a composition.

In Section 2, we describe our feature composition method. In Section 3, an event-based interaction mechanism is introduced to crosscut features. Section 4 presents the related works. Section 5 discusses some famous related works in the area of feature composition and interaction along with a running example. In Section 6, future work is discussed and Section 7 concludes the paper.

2. Feature composition

In our approach, instead of collecting features of an object in a class, each of them is defined as a separate class. Therefore, a feature becomes an independent and instantiable class. Its services can be utilized just by instantiating an object from it.

Although a feature is a class, it can also be composed with other features (classes). A composition happens when one needs to create an object having some different features. The declaration of a feature composition for object instantiation follows this syntax:

```
(F1 & ... & Fn] obj=new [F1 () & ... & Fn ()];
```

Different compositions of features cause object variations. The order of features in a composition does not matter at all since there is no hierarchy or dependency among them. In addition, each feature has its own constructor, which is executed when the object is instantiated. For example, Figure 3a instantiates each features separately and Figure 3b creates an object instance by composing both Queue and Counter features.

The most important benefits of such a composition are that:

- Roles of objects are gathered by composing features, which makes it possible to have object variations.

```
Queue q=new Queue();
Counter c=new Counter();
```

(a) Feature Instantiation

```
[Queue & Counter] qc=new [Queue() & Counter()];
qc.enqu(1);
qc.inc();
```

(b) Feature composition

Figure 3. (a) instantiates and (b) composes Queue and Counter features.

- Features, as loosely coupled reusable entities, can be simply added or removed in many objects, lowering code duplication.

- Participating a feature in any composition does not require any modification of its code.

Furthermore, multiple inheritance can be implemented naturally using our composition mechanism. Features in a composition exactly are the inherited functionalities of super classes. However, the diamond problem can also happen in our model. In fact, when two different features of an object have the same methods, calling the method name by the object causes confusion (which method of which feature must be called?). For example, in Figures 4a and 4b, both F1 and F2 features have their own method m. When we create an object, having both F1 and F2 features, and call m method without specifying the feature name, a confliction happens.

```
class F1{          class F2{          [F1 & F2] obj=new [F1() & F2()];
  void m(){...}      void m(){...}    obj->F1.m();
}                  }                  obj->F2.m();
```

(a) (b) (c)

Figure 4. In (c), a confliction over method call is resolved.

Our composition mechanism has an option for this confliction (Figure 4c). It allows the programmer to access features through objects by feature name (i.e. object–>feature.method). This way, an object accesses to the desired feature and decides on the required m method.

3. Feature interaction

This section discusses how features interact in a composition. In our model, features in a composition are encapsulated. Therefore, they cannot crosscut each other to have interactions. In fact, methods of a feature are only accessible by the composer and via object name. For example, in the composition [Queue & Counter], Queue and Counter cannot crosscut each other, e.g., enqu method of Queue cannot call inc method of Counter and vice versa. To break this restriction and allow interoperability of features, we use event as a feature interaction mechanism.

We think that every feature method reaches some specific states from the beginning to the end point of its code. The number of states a method has is limited to the size of the method (the type of work it does). When a method reaches a state, in a feature composition, other features must be notified by raising an event.

On designing a feature, a programmer has to define the events of feature methods. The definition of an event begins with the *event* keyword. Similar to a method definition, an event has a return type that can be any data type (*void* is valid too). If the return type is not *void*, it must have a default value. Essentially, an event does not have any body at all. The following code snippet shows the general form of event definition and the way of raising it.

$class \quad FeatureName\{$

$event \quad returnType \quad eventName(\overline{paramType \quad paramName}) = defaultValue;$

$... \quad method(...)\{$

$...$

$riase \quad eventName(\overline{values});$

$...$

$\}$

$\}$

An event as a part of a feature code refers to a state of a feature method. Therefore, it should have a meaningful name since it is important for crosscutting features and also helps the understandability of the feature code. For instance, BeforeAdd and AfterAdd are mostly reached events (states) in an element addition method (e.g., enqu, push, insert, etc.) of data structures (like Queue, Stack, Tree, etc).

In comparison with AOP, event raisings are not limited to the before and after points of methods. In fact, an event can be raised at any desired point of code. For example, in Figure 5, Stack at the beginning state of adding an element raises an event name evBeforeAdd and after successful adding of an element raises evAfterAdd. Moreover, two other events (i.e. evBeforeRemove and evAfterRemove) are defined and raised in pop method. These events are adequate for most of compositions in which Stack participates.

```
class Stack{
  event bool evBeforeAdd()=true;
  event void evAfterAdd();
  event bool evBeforeRemove()=true;
  event void evAfterRemove();
  void push(int value){
    if (raise evBeforeAdd()){
      ...
      raise evAfterAdd();
    }
  }
  void pop(){
    if (raise evBeforeRemove()){
      ...
      raise evAfterRemove();
      return ...
    }
  }
}
```

Figure 5. Interactive Stack feature.

An event can be taken either by a feature or not at all by any feature. On the one hand, by accepting an event, the receiving feature executes a method (in response to the event) and returns a result if needed. The result depends on the event definition. On the other hand, when an event is raised and not taken by any feature, its default value is replaced in the raise locations.

In our model, a feature should not be aware of the future features it will be composed with. It is a task

of the composing feature to compose features and delegate the events to the methods (like wiring of hardware components). For example, Stack has no information about who will catch evAfterAdd event (Counter or Log or any other feature). The important thing is that push method has reached a state named evAfterAdd. This means, it successfully added an element to the stack and this stage is the best place for another feature to crosscut Stack and do an action.

The fate of events will be determined at feature composition (object instantiation) inside composer. This means, when features are composed and an object is instantiated, it becomes clear that which event of a feature is delegated to which method of another feature. For example, Figure 6 composes Stack with Counter and just delegates two events of Stack to the Counter methods.

```
[Stack & Counter] sc = new [Stack()&Counter()]
{
  Stack.evAfterAdd=Counter.inc;
  Stack.evAfterRemove=Counter.dec;
};
```

Figure 6. Composing Stack with Counter.

An event may have some arguments depending on the state it reflexes. When a feature raises an event, it gives state values to the event arguments. For example, in Figure 7a, inc method of Counter informs other features that it is going to increase the counter by sending its value over raising of evBeforeAdd. Composer, in Figure 7c, limits Counter to 10 by composing it with Limit feature and delegating its evBeforeAdd event to the check method of Limit.

Our model allows programmers to freely define and raise events at any point of code. Moreover, there is no restriction for the arguments of events. Although our type system automatically checks for any mismatch, it is a duty of the programmer to check the signature of events and methods before any delegation.

```
class Counter{
  int cnt=0;
  event bool evBeforeAdd()=true;
  event void evAfterAdd();
  void inc(){
    if (raise evBeforeAdd(cnt))
    {
      cnt++;
      raise evAfterAdd();
    }
  }
}
```

(a)

```
class Limit{
  int size;
  public Limit(int s)
  {
    size=s;
  }
  bool check(int v)
  {
    return v<size;
  }
}
```

(b)

```
[Counter&Limit] cl
       = new [Counter()&Limit(10)]
{
  Counter.evBeforeAdd=Limit.check;
};
```

(c)

Figure 7. Passing parameters over an event.

Hierarchal relations are natural in the real world. Usually, they were modeled in programming languages by inheritance. In our approach, to model hierarchal relations, we have hierarchal composition. A hierarchal composition models one or more hierarchal relations. For example, there is a relation (IS-A) between Employee and Person features. Moreover, as Manager is an Employee, a hierarchal relation exists between them. Figure 8 illustrates these relations by hierarchal compositions.

Actually, it not essential to create a new data type for each feature composition. However, for the sake of reusability, readability, and maintainability, the programmer can make a composite feature type, which is composed of other features. When two or more features are mostly used together, it is optimal to compose

```
class EmployeePerson:[Employee & Person]
{
  public EmployeePerson(){
    Person();
    Employee();
  }
}
```

(a)

```
class ManagerEmployee:[Manager & EmployeePerson]
{
  public ManagerEmployee(){
    EmployeePerson();
    Manager();
  }
}
```

(b)

Figure 8. Modeling hierarchal relations.

them once, and reuse many times. For example, in Figure 9a, QueueLock is a new feature which is composed of Queue and Lock. It is reused in two compositions in Figure 9b. As we can see from the example, QueueLock can easily be composed with Logger and Counter.

```
class QueueLock : [Queue & Lock]
{
  Queue.evBeforeAdd=Lock.is_unlocked;
  Queue.evBeforeRemove=Lock.is_unlocked;
  Public QueueCounter(){
    Queue();
    Lock();
  }
}
```

(a)

```
[QueueLock&Counter] qlc=new [QueueLock()&Counter()]
{
  QueueLock.evAfterAdd=Counter.inc;
  QueueLock.evAfterRemove=Counter.dec;
};

[QueueLock&Logger] qlg=new [QueueLock()&Logger()]
{
  QueueLock.evAfterAdd=Logger.Log;
  QueueLock.evAfterRemove=Logger.Log;
};
```

(b)

Figure 9. Reusing feature composition.

It is obvious that hard-coding the feature interactions inside the feature definition makes it unreusable. However, our events as interactions are soft dependencies. This means that when a feature is instantiated alone (not participated in a composition), its events become neutral operations, and wherever they are raised, their default values are replaced. As a result, not only does our interaction mechanism makes features interactive, which are able to crosscut each other, but it also keeps reusability.

4. Related work

4.1. Object-oriented programming (OOP)

Inheritance is a built-in mechanism for statically refining classes in object-oriented languages. A feature of a class is encapsulated by a subclass, which can add new methods and data members, as well as override existing methods of its superclass. The variations are single and multiple inheritances.

Aggregation is another technique of OOP for reusing class features. In this model, there is a whole/part relationship between two classes (“has-a”). A synonym for this is ”part-of”. Therefore, an aggregate object is the one which contains other objects.

4.2. Mixin

A mixin is a fragment of a class in the sense that it is intended to be composed with other classes or mixins. The term mixin (or mixin class) was originally introduced by Flavors [14], the predecessor of CLOS [10]. One possibility to model mixins in object-oriented languages is to use classes and multiple inheritance. In this model, a mixin is represented as a class, which is then referred to as a mixin class, and we derive a composed class from

a number of mixin classes using multiple inheritance. Another possibility is to use parameterized inheritance. In this case, we can represent a mixin as a class template derived from its parameter, e.g.:

```
templete <class super> class derived: public super{...}
```

Indeed, some authors (e.g., [4]) define mixins as "abstract subclasses" (i.e. subclasses without a concrete superclass). Mixins based on parameterized inheritance in C++ have been used to implement highly configurable collaboration-based and layered designs (e.g., see [23, 26])

4.3. Aspect-oriented programming (AOP)

AOP was first introduced in [11] by Gregor Kickzales as an additional patch to the object-oriented software design to reach, modify, and extend the component code of system software without changing any building blocks in the system structure. The main principle of AOP is separating the nonfunctional code fragments (concerns) from the actual business logic in a modular fashion, which has not been solved in OOP. Separating the nonfunctional areas from actual business logic increases the readability, maintainability, and modularity of code. Nonfunctional code areas to business logic are often referred to as crosscutting concerns in AOP terminology.

Multidimensional separation of concerns [17, 24] is another technology for refining programs. In this model, a hyperslice is a feature, and a composition of hyperslices forms a hypermodule.

4.4. Feature-oriented programming (FOP)

FOP is an approach to modularize software according to the features it provides [25]. A feature is an increment in a program functionality [1]. The feature extensively was studied in the domain of telephony. Zave and Jackson [9, 27, 28] defined telephony features and their interface properties independently in formal description languages. This domain motivated Prehofer [18–20] [21] to develop a generalization of mixin inheritance for handling feature interactions. As a difference of mixin, he considered interactions and separated a feature from interaction handling. His work mostly focuses on feature interaction, through explicit entities (called lifters) that determine how two features interact. A lifter is a set of code modifications that is applied when its associated interaction occurs in a feature composition.

4.5. Refinement

A "refinement" is a functionality addition to a software project that can affect multiple dispersed implementation entities (functions, classes, etc.). Smaragdakis in [22] examines large-scale refinements in terms of a fundamental object-oriented technique called collaboration-based design, then explains how collaborations can be expressed in existing programming languages or can be supported with new language constructs, and at last presents a specific expression of large-scale refinements called mixin layers, which were originally inspired by the GenVoca model [3].

GenVoca is a layered design methodology for creating application families and architecturally extensible software, i.e. software that is customizable via module additions and removals. GenVoca advocates that a domain be decomposed in terms of largely orthogonal features which are implemented as layers. Applications in the domain can be synthesized by composing layers; layer composition is performed by a generator. Authors in [2] showed scaling of step-wise refinement. It introduced the AHEAD model which synthesizes multiple programs and multiple noncode representations.

4.6. Object teams

An object-oriented language with an explicit support of roles is Object Teams/Java (OT/J) [8]. Object teams aims to support the collaboration of objects, and therefore introduces two new types of class modules: roles and teams. Roles feature two special relationships. In the first relationship, a role is played by a base. A role class defines another class to be its base (via playedBy binding). Every runtime instance of the role class is associated with a corresponding instance of the base class. Base classes do not require any changes and are unaware of the adaption performed by a role. The relationship between the role and the base has many similarities with inheritance [13].

5. Discussion

In this section, we implement the composition of features in some famous approaches by a running example and discuss them from four points of view. In addition, a comparison is made between our work and theirs, focusing on how they deal with the following concerns:

1. *Instantiability*: the ability to create instances (objects) from a feature,
2. *Reusability*: the ability to use a feature in different objects,
3. *Loosely coupled composability*: the ability to compose features at object instantiation time (no new data type is required for any feature composition), and
4. *Interactivity*: the ability of a feature to crosscut (interact with) other features inside a composition.

As a running example, consider Queue and Stack classes and Counter and Lock features from Figure 1.

5.1. The proposed approach

In our model, Queue, Stack, Counter, and Lock classes are separate features. As features are instantiable, objects can be yielded just by instantiating them. The difference between the definition of Queue and Stack in our model and OOP is that our features are composable and interactive at the time of object instantiation.

In Figures 5 and 10, Queue and Stack have four events. The meaning of events in Queue and Stack are the same. These events do not affect the reusability of Queue and Stack. For example, in the case that Queue is instantiated alone or composed only with Counter, the compiler automatically removes its uncaptured events (like *evBeforeAdd* and *evBeforeRemove*) from the final object and replaces *true* (its default value) in all of its raising locations (i.e. the first statements of enqu and deque).

In our model, features are also reusable in different compositions. For example, Counter and Lock can easily be used for both Queue and Stack (see Figure 11a and 11b). Moreover, Logger feature is used instead of Counter in both compositions (see Figure 11c and 11d).

Finally, it is not essential to create an extra data type for each composition, since feature compositions happen at object instantiation time.

5.2. Aggregation

In aggregation, a feature as a class is instantiable. An object can be created from each feature. However, features cannot be composed at object instantiation time. Therefore, we cannot create a single object by composing

```
class Queue{
  event bool evBeforeAdd()=true;
  event void evAfterAdd();
  event bool evBeforeRemove()=true;
  event void evAfterRemove();
  void enqu(int value){
    if (raise evBeforeAdd()){
      ...
      raise evAfterAdd();
    }
  }
  void dequ(){
    if (raise evBeforeRemove()){
      ...
      raise evAfterRemove();
      return ...
    }
  }
}
```

Figure 10. Interactive Queue feature.

```
[Queue & Lock & Counter] qlc=
    new [Queue() & Lock() & Counter()]
{
  Queue.evBeforeAdd=Lock.is_unlocked;
  Queue.evAfterAdd=Counter.inc;
  Queue.evBeforeRemove=Lock.is_unlocked;
  Queue.evAfterRemove=Counter.dec;
};
```

(a)

```
[Stack & Lock & Counter] slc =
        new [Stack()&Lock()&Counter()]
{
  Stack.evBeforeAdd=Lock.is_unlocked;
  Stack.evAfterAdd=Counter.inc;
  Stack.evBeforeRemove=Lock.is_unlocked;
  Stack.evAfterRemove=Counter.dec;
};
```

(b)

```
[Queue & Lock & Logger] qlg=
    new [Queue() & Lock() & Logger()]
{
  Queue.evBeforeAdd=Lock.is_unlocked;
  Queue.evAfterAdd=Logger.Log;
  Queue.evBeforeRemove=Lock.is_unlocked;
  Queue.evAfterRemove=Logger.Log;
};
```

(c)

```
[Stack & Lock & Logger] slg=
    new [Stack() & Lock() & Logger()]
{
  Stack.evBeforeAdd=Lock.is_unlocked;
  Stack.evAfterAdd=Logger.Log;
  Stack.evBeforeRemove=Lock.is_unlocked;
  Stack.evAfterRemove=Logger.Log;
};
```

(d)

Figure 11. Reusing interactive features.

multiple features. Instead, for a feature composition, it is necessary to build a new data type (causing code duplication) and create an object from it. Especially when the programmer wants to make different variations of objects from a set of features, he must create a new data type (i.e. class) for each variation. As an example, when composing Lock, Counter and Queue or composing Lock, Logger, and Queue features, two new data types LockCounterQueue (Figure 12a) and LockLoggerQueue (Figure 12b) are created for each variation having three object instances inside. Code duplication is limited to object instantiations, method redefinitions and callings of feature methods. The same thing happens for composing Lock with Queue, etc. This exponentially increases the number of data types, that is, for n features, 2^n data types are possible.

The important thing about aggregation is that it does not support crosscutting concerns. This means, features cannot crosscut each other at all. In our model, features can crosscut each other by events.

```
class LockCounterQueue{
  Queue q=new Queue();
  Counter c=new Counter();
  Lock l=new Lock();
  void enqu(int value){
    if (l.is_unlocked()){
      q.enqu(value);
      c.inc();
    }
  }
}
```

(a)

```
class LockLoggerQueue {
  Queue q=new Queue();
  Logger g=new Logger();
  Lock l=new Lock();
  void enqu(int value){
    if (l.is_unlocked()){
      q.enqu(value);
      g.Log();
    }
  }
}
```

(b)

```
class LockQueue{
  Queue q=new Queue();
  Lock l=new Lock();
  void enqu(int value){
    if (l.is_unlocked())
      q.enqu(value);
  }
}
```

(c)

Figure 12. Composing features by aggregation.

5.3. Single inheritance

In single inheritance, features as subclasses are placed in the inheritance hierarchy. Each feature contains the functionalities of its super classes. The order in which features are placed in the hierarchy is important. See the following cases.

- LockCQ inherits CounterQ which inherits Queue
- CounterLQ inherits LockQ which inherits Queue

In the first feature composition, objects created from CounterQ involve the queue and counting functionalities. But in the second composition model, objects of CounterLQ have functionalities of queue and lock in addition to counting. Notice that it is not possible to create a pure Counter object in both hierarchy models. To do so, it is required to create another data type (i.e. Counter class) which does not inherit any class. This leads to code duplication. As a result, features are instantiable but creating pure objects from features usually requires building new data types.

Single inheritance does not provide feature reusability. For instance, CounterQ (as a feature of Queue) cannot be used for Stack. Therefore, another identical Counter class (e.g., CounterS) must be created. Again, this causes code duplication. Thus, instead of reusing a feature a new data type must be created.

```
class CounterQ:Queue{
  int cnt=0;
  void inc(){cnt++;}
  void dec(){cnt--;}
  int size(){return cnt;}

  void enqu(int value){
    super.enqu(value);
    inc();
  }
  int dequ(){
    dec();
    return super.dequ();
  }
}
```

(a)

```
class CounterS:Stack{
  int cnt=0;
  void inc(){cnt++;}
  void dec(){cnt--;}
  int size(){return cnt;}

  void push(int value){
    super.push(value);
    inc();
  }
  int pop(){
    dec();
    return super.pop();
  }
}
```

(b)

Figure 13. Feature composition by single inheritance.

Furthermore, features are semiinteractive in the normal form of single inheritance. This means, only child features can interact with their parent features. However, a parent feature can be interactive by using polymorphism. This is shown by an example. In Figure 14, Queue has an empty implementation of inc method, since it has an interaction with CounterQ by calling inc. In the case that enq is called by a Queue object, executing its last statement will invoke inc method of Queue and accordingly calling enq by a CounterQ object will invoke inc method of CounterQ. In such a way, Queue and CounterQ can have interactions with each other, Queue by calling inc and CounterQ by calling enq.

```
class Queue{
  void enqu(int value){
    ...
    inc();
  }
  virtual void inc(){}
}
```

(a)

```
class CounterQ:Queue{
  int cnt=0;
  void inc(){cnt++;}
}
```

(b)

Figure 14. Making parent feature interactive by polymorphism.

There are three important differences between such an interaction and our event-based interaction mechanism. First, this model follows name matching, that is, the name of interacting method (e.g. inc) must be the same in both parent and child features. Whereas our interaction mechanism breaks this restriction and follows name mapping; event and method names could be different.

Second, a parent feature must have an empty implementation of all interacting methods. This increases the size of the feature code and imposes code duplication. Moreover, it is a tedious task. While a feature, in our model, does not provide any implementation for its own events.

Third, in single inheritance, interactions are hard dependencies. This means, always, there is an overhead over calling a method for each interaction. For example, inside enq method of Queue, an inc call always exists. In contrast, our interactions are soft dependencies. An event is not always translated into a method and does not impose an overhead. In the case that a feature does not participate in an anticipated composition, some of its events are not captured. Raising an uncaptured event is not a method call anymore. It is finally evaluated to its default value.

5.4. Multiple inheritance

In the multiple inheritance, a feature is an instantiable class. A feature composition is done by introducing a new data type, which inherits some classes. Figure 15 shows two different feature compositions in C++ multiple inheritance result in two different new data types. Both have Counter and Lock features in common.

In this model, a feature can be reused in many compositions (e.g. Counter is used in both Stack and Queue compositions). However, features are not interactive in compositions. The interactions of features in a composition are managed by the new derived class. For instance, the interactions between Queue and Counter are managed by the QueueCounterLock class.

5.5. Mixin inheritance

Mixin inheritance is similar to single inheritance except that a mixin class can have different super classes. This makes a mixin reusable in different compositions. However, by looking at the interactions of child mixins with their super classes, we understand that special super-classes are acceptable. This is shown by an example. In

```
class QueueCounterLock
  :public Queue,
  public Counter,
  public Lock
{
public:
QueueCounterLock()
  :Queue(),Counter(),Lock(){}

  void enqu(int value){
    if (is_unlocked()){
      Queue::enqu(value);
      inc();
  }
};
```

(a)

```
class StackCounterLock
  :public Stack,
  public Counter,
  public Lock
{
public:
StackCounterLock()
  :Stack(),Counter(),Lock(){}

  void push(int value){
    if (is_unlocked()){
      Stack::push(value);
      inc();
  }
};
```

(b)

Figure 15. Multiple inheritance: feature compositions and interactions.

Figure 16, since interactions of Queue and Stack mixins are through calling inc and dec methods, their supers must implement these methods. This means, Queue and Stack are reusable just in the presence of a super which provides inc and dec methods.

Moreover, a mixin as a feature is instantiable just in the presence of its super-class. For instance, to create pure objects from Queue and Stack mixins, either an empty super mixin or another Queue and Stack mixins (which do not have any interactions) must be created.

```
template <class Super>
class Queue:public Super{
public:
  void enqu(int value){
    Super::inc();
    ...
  }
  int dequ(){
    Super::dec();
    ...
  }
}
```

(a)

```
template <class Super>
class Stack:public Super{
public:
  void push(int value){
    Super::inc();
    ...
  }
  int pop(){
    Super::dec();
    ...
  }
};
```

(b)

```
Queue<Counter> qc;
Stack<Counter> sc;
```

(c)

Figure 16. Queue and Stack features in mixin.

Finally, a mixin composition does not need to create a new data type. Moreover, like single inheritance, features are interactive with the cost of reusability.

5.6. Lifters in FOP

In FOP, any functionality is placed in a structure called feature. Like a class, a feature is instantiable and implements an interface. For example, QF, SF, CF, and LF are features which implement Queue, Stack, Counter, and Lock functionalities, respectively (Figure 17).

To create a single object, like our model, features can be composed with each other at object instantiation time. Therefore, object variation is reachable. See the following code snippet.

```
LF(CF(QF)) lcq=new LF(CF(QF));
LF(CF(SF)) lcS=new LF(CF(SF));
```

```
feature QF
    implements IQueue
{
  void enq(int value)
  {...}
  int deq() {...}
}
```
(a)

```
feature CF
    implements ICounter{
  int cnt = 0;
  void inc(){cnt++;};
  void dec(){cnt--;};
  int size(){return cnt;};
}
```
(b)

```
feature LF
    implements ILock
{
  bool l=true;
  void lock(){l=false;}
  void unlock(){l=true;}
  bool is_unlocked()
  {return l;}
}
```
(c)

```
feature SF
    implements IStack
{
  void push(int value)
  {...}
  int pop(){...}
}
```
(d)

Figure 17. Features in FOP.

However, features cannot crosscut each other. To manage feature interactions, for any possible compositions of features, there is a unique Lifter. However, a Lifter is not a new data type. It just manages the interaction of two distinct features (Figure 18).

```
feature CF lifts IQueue{
  void enqu(int v)
  {
    this.inc();
    super.enq(v);
  }
  int dequ()
  {
    this.dec();
    return super.deq();
  }
}
```
(a)

```
feature CF lifts IStack{
  void push(int v)
  {
    this.inc();
    super.push(v);
  }
  int pop()
  {
    this.dec();
    return super.pop();
  }
}
```
(b)

```
feature LF lifts ICounter {
  void inc() {
    if (this.is_unlocked())
      { super.inc(); }
  }
  void dec() {
    if (this.is_unlocked())
      { super.dec(); }
  }
}
```
(c)

Figure 18. Managing feature interactions by Lifters

5.7. Aspect-oriented programming (AOP)

In AOP, an aspect is not instantiated. Therefore, it is not possible to create objects from it. Besides, aspects as features are dependent on special classes. They cannot be composed with each other without classes. Such dependency influences aspect reusability.

For example, in Figure 19, it is not possible to create an object from CounterQueue Aspect. This aspect is defined just to add the counting feature to Queue class. Therefore, it cannot be used for Stack. To add counting feature to Stack, it is required to make another aspect (e.g. CounterStack). As a result, aspects cause code duplication and do not provide reusability.

Furthermore, aspects are per class rather than per object, that is, they affect class definitions. This means

```
public aspect CounterQueue{
  int Queue.cnt=0;
  pointcut callenqu()
    : call(* Queue.enqu(*));
  pointcut calldequ()
    : call(* Queue.dequ(*));

  after():callenqu(){cnt++;}
  after():calldequ(){cnt--;}
}
```

(a)

```
public aspect CounterStack{
  int Stack.cnt=0;
  pointcut callpush()
    : call(* Stack.push(*));
  pointcut callpop()
    : call(* Stack.pop(*));

  after():callpush(){cnt++;}
  after():callpop(){cnt--;}
}
```

(b)

Figure 19. Counting Aspects over Queue and Stack.

that all the objects created from a class will have the effect of its aspects. Consequently, aspect composition does not provide object variations.

Another important defect is that an aspect as a feature cannot change the control flow of its base class. This is critical when we have conditional cases. For example, in Lock aspect (Figure 20), it is required to inject lock check by *if* command before Queue/Stack insertion. This is impossible with aspects while we do it in our model.

```
public aspect Lock {
  bool l=true;
  void lock(){l=false;}
  void unlock(){l=true;}
  bool is_unlocked(){return l;}

  pointcut callAllMethods(): call(* *.*(*));
  before(): callAllMethods(){
    if (is_unlocked()){
      ***
    }
  }
}
```

Figure 20. Lock Aspect

5.8. Step-wise refinement

In step-wise refinement, a refinement as a feature of a class is not instantiable since it is an incomplete entity. It is defined only for a specific class, not reusable for other classes. For example, in Figure 21, refinements of Stack are neither instantiable nor reusable for Stack. The same thing happens for the refinements of Queue. It is important to know that composing refinements of a class does not make new data types. Besides, refinements are not interactive.

Table evaluates our model in comparison with related works. It is clear that reusability and interactability are opposite criteria. However, our interaction method does not affect reusability.

6. Future work

Future research should consider the potential effects of events more carefully in the design patterns problems and also domain of collaboration-based design. Real-world examples should be taken into account in order to examine the advantages and disadvantages of the approach. In the next work, we will work on a new role-oriented programming approach and bring events into the collaborative component.

```
refines Queue {
  int cnt=0;
  void inc(){cnt++;}
  void dec(){cnt--;}
  int size(){return cnt;}
  void enqu(int value){
    super.enqu(value);
    inc();
  }
  int dequ(int value){
    dec();
    return super.dequ(value);
  }
}
```

(a)

```
refines Queue {
  bool l=true;
  void lock(){l=false;}
  void unlock(){l=true;}
  bool is_unlocked()
    {return l;}
  void enqu(int value){
    if (is_unlocked())
      super.enqu(value);
  }
  int dequ(){
    if (is_unlocked())
    return super.dequ();
  }
}
```

(b)

```
refines Stack {
  int cnt=0;
  void inc(){cnt++;}
  void dec(){cnt--;}
  int size()
    {return cnt;}
  void push(int value){
    super.push(value);
    inc();
  }
  int pop(){
    dec();
    return
     super.pop(value);
  }
}
```

(c)

```
refines Stack {
  bool l=true;
  void lock(){l=false;}
  void unlock()
    {l=true;}
  bool is_unlocked()
    {return l;}
  void push(int value){
    if (is_unlocked())
      super.push(value);
  }
  int pop(){
    if (is_unlocked())
    return
      super.pop();
  }
}
```

(d)

Figure 21. Refining Queue and Stack.

Table. Comparing our approach with the existing approaches.

	Instantiable features	Reusable features in compositions	Loosely coupled composability	Crosscutting (Interactive) features
Aggregation	✓	✓	χ	χ
Single inheritance	✓	χ	χ	✓
Multiple inheritance	✓	✓	χ	χ
Mixin inheritance	χ	χ	χ	✓
Lifters of FOP	✓	✓	✓	χ
AOP	χ	χ	✓	χ
Refinement	χ	χ	✓	χ
Our model	✓	✓	✓	✓

By the way, we will discuss design pattern [7, 12, 15] problems (like Decorator and Template Method) in a separate work and compare our approach with the current solutions (such as GoF), in terms of understandability, flexibility, and reusability.

Furthermore, feature, event, and delegation are also needed to be visualized. Features can be modeled by extending the UML class diagram notation and delegations during feature composition can be modeled by extending the UML sequence diagram notation. Raise points also need to be marked in both or one of those diagram types.

7. Conclusions

This paper proposes a new model of feature programming which has two new methods: object composition and feature interaction. Additionally, a comparison is done between our model and existing works (like OOP, AOP, and FOP) based on four criteria: instantiability, reusability, loosely coupled composability, and interactibility. The benefits gained from this paper are: 1) A feature either can be instantiated alone or participates in compositions. This means a feature can be a whole object or part of a big object. 2) By using our composition method, programmers have object variations. This is due to the fact that features are loosely coupled independent entities, which can be easily added/removed to/from objects at object instantiation time. 3) Against inheritance-based models, in our model, it is not needed to create a new data type for each composition. This reduces the number of data types in the library. 4) Our event-based feature interaction mechanism overcomes the hard dependency between child and parent classes in OOP. Unlike a method call, an event raising is a soft dependency, which is active when a special feature has come in the composition, otherwise it is inactive.

Finally, the comparison demonstrates the existing works could not provide reusability and interactability simultaneously. The experiment results in Table 1 show that the proposed model makes classes instantiable, composable, reusable, and interactive altogether that is the most important outcome of this paper.

References

[1] Abilio R, Vale G, Figueiredo E, Costa H. Metrics for feature-oriented programming. In: Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics, WETSoM '16; New York, NY, USA; 2016. pp. 36-42.

[2] Batory D, Sarvela J N, Rauschmayer A. Scaling step-wise refinement. IEEE Transactions on Software Engineering 2004; 30 (6): 355-371. doi: 10.1109/TSE.2004.23

[3] Batory D, O'Malley S. The design and implementation of hierarchical software systems with reusable components. ACM Transactions on Software Engineering and Methodology (TOSEM) 1992; 1 (4): 355-398. doi: 10.1145/136586.13658

[4] Bracha G, Cook W. Mixin-based inheritance. SIGPLAN Not. 1990; 25 (10): 303-311. doi: 10.1145/97946.97982

[5] Duggan D, Techaubol CC. Modular mixin-based inheritance for application frameworks. SIGPLAN Not. 2001; 36 (11): 223-240. doi: 10.1145/504311.504299

[6] Flatt M, Krishnamurthi S, Felleisen M. Classes and mixins. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98; New York, NY, USA; 1998. pp. 171-183.

[7] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-oriented Software. Boston, MA, USA: Addison-Wesley, 1995.

[8] Herrmann S. A precise model for contextual roles: the programming language ObjectTeams/Java. Applied Ontology 2007; 2 (2): 181-207.

[9] Jackson M, Zave P. Distributed feature composition: a virtual architecture for telecommunications services. IEEE Transactions on Software Engineering 1998; 24 (10): 831-847. DOI: 10.1109/32.729683

[10] Keene SE. A Programmer's Guide to Object-oriented Programming in Common LISP. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 1988.

[11] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C et al. Aspect-oriented programming. In: ECOOP'97 — Object-Oriented Programming; Berlin, Heidelberg; 1997. pp. 220-242.

[12] Maniyath J. Design patterns generic models. In: Proceedings of the 18th Conference on Pattern Languages of Programs, PLoP '11; New York, NY, USA; 2011, pp. 14:1-14:5.

[13] Mertgen A. Decoupling context: Introducing quantification in object teams. In: Proceedings of the 2012 Workshop on Modularity in Systems Software, MISS '12; New York, NY, USA; 2012. pp. 45-50.

[14] Moon DA. Object-oriented programming with flavors. In: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications, OOPSLA '86; New York, NY, USA; 1986. pp. 1-8.

[15] Niculescu V. Mixdecorator: An enhanced version of decorator pattern. In: Proceedings of the 20th European Conference on Pattern Languages of Programs, EuroPLoP '15; Kaufbeuren, Germany; 2015. pp. 36:1-36:12.

[16] Nystrom N, Qi X, Myers AC. J&: Nested intersection for scalable software composition. SIGPLAN Not. 2006; 41 (10): 21-36. doi: 10.1145/1167515.1167476

[17] Ossher H, Tarr P. Using multidimensional separation of concerns to (re)shape evolving software. Communications of the ACM 2001; 44 (10): 43-50. doi: 10.1145/383845.383856

[18] Prehofer C. From inheritance to feature interaction or composing monads. In: Informatik '97 Informatik als Innovationsmotor; Berlin, Heidelberg; 1997. pp. 562-571.

[19] Prehofer C. Feature-oriented programming: A fresh look at objects. In: ECOOP'97— Object-Oriented Programming; Berlin, Heidelberg; 1997. pp. 419-443.

[20] Prehofer C. An object-oriented approach to feature interaction. In: Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems; Hyderabad, India; 1997. pp. 313-325.

[21] Prehofer C. Feature-oriented programming: a new way of object composition. Concurrency and Computation: Practice and Experience 2001; 13 (6): 465-501. doi: 10.1002/cpe.583

[22] Smaragdakis Y, Batory D. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. ACM Transactions on Software Engineering and Methodology (TOSEM) 2002; 11 (2): 215-255. doi: 10.1145/505145.505148

[23] Smaragdakis Y, Batory DS. Implementing layered designs with mixin layers. In: Proceedings of the 12th European Conference on Object-Oriented Programming, ECCOP '98; Berlin, Heidelberg; 1998. pp. 550-570.

[24] Tarr P, Ossher H, Harrison W, Sutton, Jr. SM. N degrees of separation: Multi-dimensional separation of concerns. In: Proceedings of the 21st International Conference on Software Engineering, ICSE '99; New York, NY, USA; 1999. pp. 107-119.

[25] Thüm T, Apel S, Zelend A, Schröter R, Möller B. Subclack: feature-oriented programming with behavioral feature interfaces. In: Proceedings of the 5th Workshop on MechAnisms for Specialization, Generalization and inHerItance, MASPEGHI '13; New York, NY, USA; 2013. pp. 1-8.

[26] VanHilst M, Notkin D. Using role components in implement collaboration-based designs. SIGPLAN Not. 1996; 31 (10): 359-369. doi: 10.1145/236338.236375

[27] Zave P. Feature interactions and formal specifications in telecommunications. IEEE Computer 1993; 26 (8): 20-30. doi: 10.1109/2.223539

[28] Zave P. An experiment in feature engineering. In: McIver A and Morgan C (editors). Programming Methodology. New York, NY, USA: Springer New York, 2003, pp. 353-377.