Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelor thesis

Marvin Ede

Increasing modularity by implementing traits in ruby with an application to game programming

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Marvin Ede

Increasing modularity by implement traits in ruby with an application to game programing

Bachelorarbeit eingereicht im Rahmen Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Michael Böhm
Zweitgutachter : Prof. Dr. Ing. Birgit Wendholt

Abgegeben am 27.03.2013

Marvin Ede

Thema der Bachelorarbeit

Erhöhte Modularität durch die Implementierung von traits in ruby mit einer Anwendung auf Spieleprogrammierung

Stichworte

Modularität, Ruby, Trait, Spieleprogrammierung

Kurzzusammenfassung

Diese Arbeit definiert Traits – feingranulare, wiederverwendbare Module, aus denen Klassen zusammengesetzt werden können. Es wird gezeigt wie mit Hilfe der Metaprogrammiertechniken aus Ruby solche Traits implementiert werden können. Dabei wird auf die bereits vorhandenen Mixins aufgebaut ohne etwas anderes als Rubysyntax zu verwenden. Die benötigten Vorkenntnisse über die Metaprogrammierung in Ruby werden in einem eigenen Kapitel vermittelt.
Nachdem die Schwachpunkte dieser Herangehensweise erläutert wurden, wird gezeigt wie man Traits bei einer Spielengine anwenden könnte und wie sie im Anwendungscode des Spiels zu mehr Modularität führen.

Marvin Ede

Title of the paper

Increasing modularity by implementing traits in ruby with an application to game programming

Keywords

Modularity, ruby, trait, game programming

Abstract

This work gives a definition for traits – a fine grained, reusable set of methods that is used to build classes. It is shown how rubies meta programming techniques can be used to implement such traits based on mixin modules only using ruby syntax. The needed meta programming techniques are described in a separate chapter. After the weaknesses of this approach and its limits are discussed, it is shown how traits could be applied to a game engine and how that would lead to more modularity in the actual game application code.

Index

1 Introduction

In the past decades of software development, there have been many changes in languages and in the way they are used. While in the grand scheme of paradigms, only the imperative and the declarative ones are established there are many different manifestations of these two approaches. For declarative programming there is distinguished among functional, logical and constraint programming, while imperative programming can be sorted as procedural, modular, agent oriented and object oriented – to name a few.
Object oriented programming experienced a significant accretion in the past decades, especially languages like Java and C++ propagate this paradigm. To face the different problems of the most common variant of object orientation – the single inheritance – there has been different approaches to extend the object model, one of which are traits.

1.1 Problem

One of the benefits from the object oriented programming paradigm is, that programs are easier to read and easier to understand, thus easier to maintain. One big problem with maintenance is duplicated code. With classical object oriented techniques like single inheritance it is shown that there are situations where code must be duplicated, which – in the long run - leads to diversion of two pieces of code, that once were designed to implement the same feature (cf. (Murphy-Hill, et al., 2005))
Traits as an extension to the object oriented programming paradigm are designed to offer solutions to this problem. However, programmers are often not even aware of those problems, nor do they know traits or other possible solutions. This may be caused by the lack of implementations of traits[1] and practical applications of traits that can function as a showcase.

[1] To the knowledge of the author, there is no implementation of traits, like they are defined in this work, for ruby at the time of publication.

1.2 Goal

In this work, there will be discussed how classic object oriented techniques are not sufficient to redeem the causes of duplicated code. It will be shown how traits are designed to solve these problems and to extend current techniques like single inheritance. The main emphasize however, lies on the analysis of how traits can be implemented using rubies[2] meta programming techniques. To give an example of how traits can be used in actual software, it is shown how a 2D game engine written in ruby could be refactored with traits. This framework is the 2D game engine Chingu[3] that is based on Gosu[4].

1.3 Textual structure

This work is in divided into four essential chapters plus the two chapters for introduction and conclusion. In chapter 2, traits are introduced. A detailed definition is given, that follows very closely to the definition given by Ducasse et al in 2003. (cf. (Ducasse, et al., 2006))

In chapter 3, there is an introduction to the object oriented programming language ruby. It is assumed the reader has knowledge of the object oriented programming paradigm, therefore only those parts of ruby – precisely some of its meta programming techniques – that are important for the proposed implementation of traits, are covered.

After that, chapter 4 combines the definition of traits and the introduction to ruby by describing the implementation of traits the author proposes. Not only is shown how traits are defined and used, it is dwelled on weaknesses of the implementation and alternatives that might solve these weaknesses are discussed.

In chapter 5 it is shown how traits can be used to increase modularity specific in game programming. The game engine Chingu is shown, that already makes use of the concept of mixin, that are similar to traits. It is shown how Chingu could be adapted to make use of the proposed trait implementation and how this would affect the actual application code.

Finally a conclusion is given in chapter 6. Additionally there is a prospect of how the implementation could be improved on.

Traits in the meaning of this work have been introduced in 2002 ((Schärli, et al., 2002)) and have not been the target of research in many instances since. Therefore there are only few researches to build upon. Due to this circumstance the list of source materials regarding traits may appear shorter than expected.

In this work, there are many illustrations for the presented concepts and implementation. They are designed by the author unless otherwise stated.

[2] Ruby programming language - http://www.ruby-lang.org

[3] Chingu - Make games with Ruby! - http://ippa.se/chingu

[4] Gosu - 2D game development library - http://www.libgosu.org/

2 Traits

Traits are an extension to the object oriented programming paradigm that is designed to increase the reusability of code - especially code that would have to be copied and pasted in an ordinary class hierarchy. The underlying problem is a conceptual one, as a class has two roles: It is a generator of instances and a unit of reuse. In order to fulfill its role as instance generator, it needs to be complete, therefore bundling all functionalities that an instance needs to operate properly. This leads to classes that are rather big collection of functions with a special purpose. But this completeness of a class obstructs its role as a unit of reuse. (cf. (Schärli, et al., 2003 p. 2))

2.1 Motivation – Why traits?

In other words: A class needs to be complete in order to generate instance. But it is hard to reuse a complete unit of code in another place. There are several approaches to solve this problem like single inheritance, multi inheritance and mixin inheritance. These approaches will be reviewed in a chronological order to show the need of traits in an evolutionary perspective. At the end of chapter 2.1 there will be a top-down view of traits that does not emphasize the evolution of traits but rather their concept itself.

2.1.1 Procedural programming

With procedural programming, there are only a few language artifacts to consider: Local and global variables, procedures, data types and iterators. A program is basically a monolithic collection of procedures, which can be called from anywhere. This emphasizes a basic technique of reusable code – the procedure, but big application can become confusing as there are no such things as name spaces or classes to structure the source code in a more conceptual way.

2.1.2 Parallel classes

With classes, methods can be sorted into different small domains to increase the overall maintainability of applications. They can also map the behavior of real world objects with a life cycle and states in a more intuitive way. Classes also help to emphasize information hiding to further decouple the modules of an application. All of these are major improvements compared to procedural programming. Although all those improvements help maintain an application, classes alone do not necessarily improve the reusability of code[5], as it is hard to implement a single point of control. In many parallel classes, there is lots of duplicated code if they model similar parts of an application.

2.1.3 Single inheritance

Single inheritance helps to overcome the problem of duplicated code among parallel classes. It enables a class to have one optional super class, which means it can access the methods of that class. Some methods can be overwritten while others are not. Methods can even be combined with the `super` keyword. When there is similar behavior of parallel classes, that behavior can be refactored into a common super class where the code is only defined one time, thus offering a single point of control. This technique already solves a lot of problems regarding duplicated code.

Single inheritance is the most common way to break a class into reusable chunks of code. According to (Schärli, et al., 2003 p. 1) it "is well-accepted, [but] it is not expressive enough to allow the programmer to factor out all the common features shared by classes in a complex hierarchy". In order to reuse a set of methods in a new class, it may be required to restructure the class hierarchy, thus possibly breaking its integrity or it may be required to copy and paste those methods. (cf. (Murphy-Hill, et al., 2005 S. 1)) Along with that, it is a common practice to place methods too high in the hierarchy only to be able to share its methods in different sub classes. The term "too high" means that those methods are not shared by all sub classes. (cf. (Schärli, et al., 2003 pp. 20,21)) This can even lead to class hierarchies, where an indirect subclass re-implements a method from its indirect super class by copieng and pasting, because the intermediate class did overwrite it. (cf. (Murphy-Hill, et al., 2005 S. 4))

The underlying problem is that it is not possible to define more than one super class for a certain class therefore it is only possible to design the class hierarchy from one point of view.

2.1.4 Multiple inheritance

With multi inheritance, a class can have more than one super class. This leads to much more flexibility in the class hierarchy as not only one point of view can influence the class hierarchy. Yet, there are still some problems that arise with this approach.

There can be conflicts in the inheritance of methods and state variables. Especially the so called diamond problem (cf. (Malayeri, 2008)), where a class has an indirect super class multiple times via different paths, can be tricky to solve – as the implementer of multi inheritance as well as the programmer using multi inheritance.

While method conflicts can be solved by overriding, state conflicts can be more troublesome. (cf. (Schärli, et al., 2003 p. 4))

[5] Of course, some classes may be used for another application, but within a single application, a class can most likely not be reused.

Accessing override features also is not trivial with multi inheritance. The simple keyword super, is not sufficient, as it is unclear which super class is meant. By directly referencing the super class a lot of flexibility is lost in terms of rearranging the class hierarchy. All explicit super class references would need to be reviewed.
In (Schärli, et al., 2003) there is described another problem with generic wrappers and multiple inheritance. An example of a wrapper that synchronizes read and write methods in two classes is given. There are two solutions presented that make use of multi inheritance, but none of which is free of duplication.
The most important characteristic of multiple inheritance regarding this work is its lack of symmetric composition. The order in which classes are aligned within the class hierarchy is important when the keyword super is used, in other words, if behavior from multiple classes is combined.

2.1.5 Mixin inheritance

Up to this point, the problem of duplicated code was solved with classes and inheritance alone. It is shown that there are still some cases in which code must duplicated even with multi inheritance quite apart from its other downfalls like complexity in implementation. The mixin inheritance introduces a new code artifact to the object oriented paradigm: The mixin. It allows the designer of a class to mix in a set of methods that are independent from the classical class hierarchy. The simplicity of single inheritance and the keyword `super` are retained, as the mixin becomes the only super class of the class that makes use of it. The original super class becomes the super class of the mixin. Actually, though, not the mixin itself is placed in the class hierarchy, but rather a so called proxy class (cf. (Perrotta, 2010 S. 26)) that wraps all the methods of the mixin.
Therefore, mixins can be used in different locations in the hierarchy.

Example for a mixin.

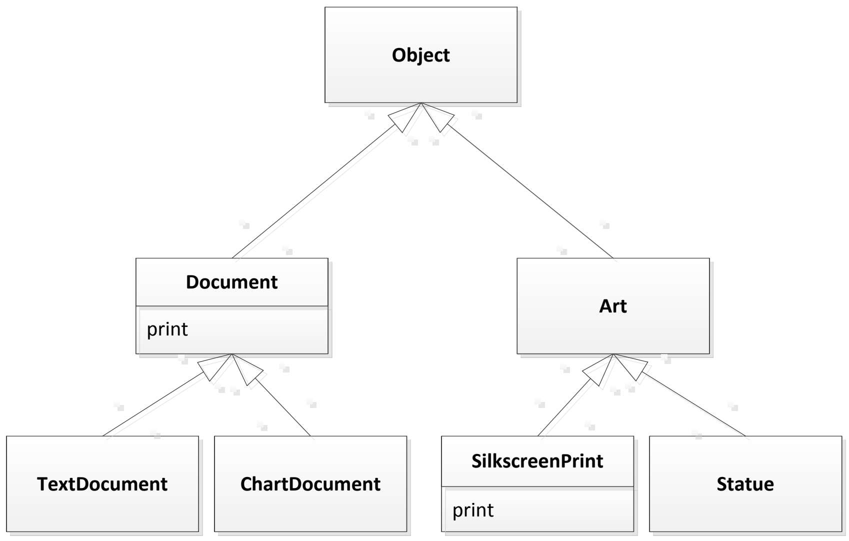


Fig 1: A class hierarchy without mixins

In this class hierarchy, there are two things that can be printed: Documents in general and silkscreen printings. Assuming that the code for printing those two things is the same, there is no legit way to share it between those two classes – its needs to be duplicated or to be placed too high in `Object`.

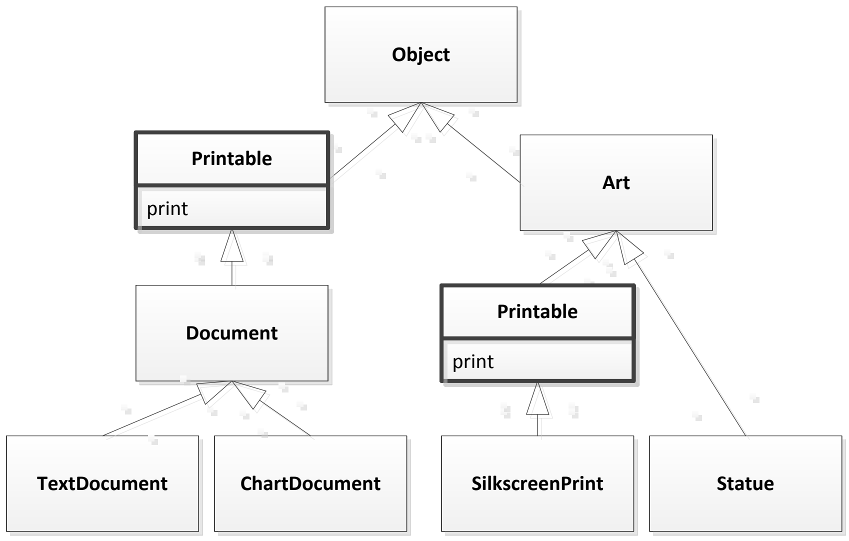


Fig 2: The same class hierarchy with mixins (Mixins are bold)

This is still the same class hierarchy, but extended with mixins. The mixin `Printable` now contains the print method, only implementing it once. It is placed in the appropriate places in the class hierarchy as it is mixed in by `Document` and by `SilkscreenPrint`.

A reusability problem with mixins arises when they are composed. A class can mix in multiple mixins, what still is unproblematic when their methods are disjoint. When multiple mixins, that are used by the same class, implement the same method, the version of the last mixin is used, as it becomes the direct super (proxy-)class of the sub class, therefore overriding all above version in mixins and real super classes alike.
This implicit overriding can be hard to catch by itself, but it gets even more messy, when the different versions of the same method need to be combined. An explicit super call in each of the versions is needed:

- Calling super can simply be forgotten. Maybe the programmer does not understand the model of a super proxy class and does not even know super could be used.
- The mixin is no longer independent from its position in the class hierarchy. What if the mixin is the last in a chain of mixins, so its super class is a real class? This class maybe does not implement the method that is being combined. Therefore the last mixin may not be allowed to call super. But then, what if it is used in another context where it is not the last mixin? It would simply stop the super chain.
  Also the programmer needs to know whether a method will be combined or not, but this depends on its position in the class hierarchy.

In essence, these problems with a chain of super calls arise, because, due to single inheritance, mixins can only be composed linearly. This means the order in which the method look up finds the different implementations of the same method is dependent from the order in which mixins where included.
A mixin can include several other mixins, therefore it is possible to have some tree-like structure of composed mixins. This however, may not be confused with the order in which the method look up or the `super` keyword finds the different implementations. The process of defining this order is called linearization.

Table 1: Evolution of programming paradigms

Evolution of programming paradigms			
Paradigm	**Short description**	**Problems**	**Solved by**
Procedural programming	Only procedures and local variables; no name spaces	Monolithic structure, a lot of duplicated code	Parallel classes
Parallel classes	Name spaces, object lifecycle	Duplicated code among similar classes	Single inheritance
Single inheritance	Common behavior can be refactored into a super class	Still duplicated code, as there is only one class hierarchy from one point of view	Multi inheritance
Multi inheritance	More than one super class per class, a class hierarchy can emphasize more than one point of view	Diamond problem, not symmetric, complexity	Mixins
Mixins	Classes can mixin methods independent from class hierarchy	Not symmetric, linear composition	Traits

2.1.6 Traits

In the evolutionary view, traits are mixins, which solve the problem of asymmetric composition. In clear words: Composition of traits is always **symmetric**. The order in which traits are incorporated into a class does not the effect the behavior of the class, as opposed to mixins. This is achieved by **explicit conflict solving**.

From the top-down view, traits in software are quite similar to traits in the real world. When an instance represents a concrete thing in the real world, its class represents the abstract idea behind a chunk of similar concrete things. For example, when a person sees a thing in a real world, which has roots, has a trunk, has branches, has leafs, is immobile and that can burn, he can call it a tree. These six habits of a concrete thing in the real world are conducted to an abstract idea of a tree. This concept of the relation between instances and classes is similar to Plato's Theory of Forms. (cf. (Hirschberger, 2007 S. 97-103))

A trait is a habit, that is involved into the Form (in software: class) of a thing in the real world (instance). It is an integral component of the Form but not only of the one Form but of other Forms as well. The Form of a tree, in this case, consists of the habits of having

roots, having a trunk, having branches, having leafs, being immobile and being burnable. All of these six habits can be translated into traits. In software, a class Tree would incorporate the six traits named above.
If the traits are fine-grained enough to satisfy the given model of the world, Forms (classes) do only consist of traits and have no habits on its own.

2.1.7 An Example for traits in the game programming domain

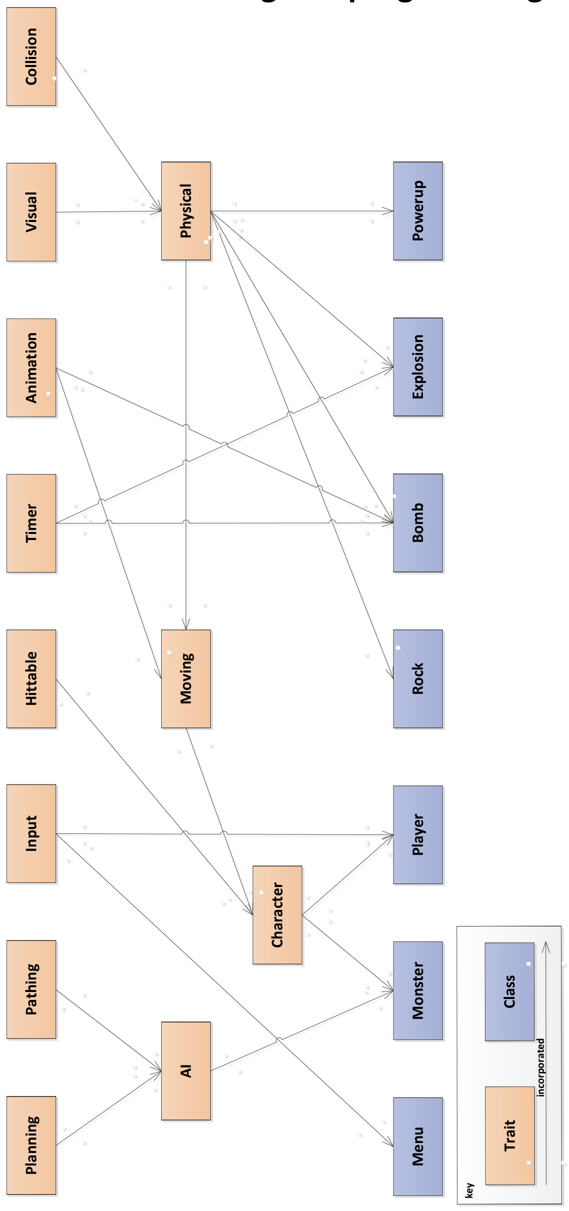


Fig 3: This is an example of a possible class hierarchy that heavily relies on traits. Common traits like animation, input or collision detection are implemented in a class but in a trait, they are then included only into those classes that need this behavior. The addition of composed traits like `Physical` or `Character` makes the definition of classes even more comfortable. An equivalent class hierarchy without those composed traits can be seen in appendix.

2.2 A definition of traits

"A trait is essentially a group of pure methods that serves as a building block for classes and is a primitive unit of code reuse." – Schaerli et al

The citation above gives a good idea of what traits are, as it goes into these three questions: What is a trait? ("a group of pure methods") What is its purpose? ("a primitive unit of code reuse") How are they used? ("…as a building block for classes") This chapter will give a more detailed view on traits.

2.2.1 Pure methods

A trait is a "group of pure methods". What does "pure" mean in this context? Even though in (Schärli, et al., 2003) this word is not explained any further, the author believes it is safe to assume it relates to the fact that "…traits do not specify any state variables, and the methods provided by traits never access state variables directly." ((Schärli, et al., 2003 p. 2))
This is done so that conflicts regarding the state of an object do not need to be considered by traits at all. The class and only the class - as an instance generator - needs to consider the state of an object. Traits apply to a more abstract view of class as a collection of higher tier functions.

2.2.2 Provided, required and satisfied methods

A trait implements a set of methods that it *provides* to the class it is incorporated into. These methods can be based on other methods that the class needs to implement. These methods are provided by the class or from the viewpoint of the trait they are required. When a class provides (implements) a method that is required by a trait, it satisfies this method. If all required methods of a trait are satisfied, the trait is satisfied.

The possibility of required methods is what makes a trait incomplete and therefore a proper unit of reuse. According to the principle of functional cohesion, the traits that a class is built from, should regard only one behavior of the class. If a trait tries to solve too many problems, it becomes once again a near-complete set of units that is hard to reuse.

Example for provided, required and satisfied methods.
A class `Player` in a game needs to have the behavior of positioning on the screen. The programmer decides it is time to define a trait as the logic of positioning something on the screen will be reused later in other classes. The behavior of positioning is based on a state of x and y. So the Player class needs to offer methods regarding that state. In this case these are `x()`, `x=(value)`, `y()` and `y=(value)`. The trait `Position` requires the

class `Player` to implement these four methods. In return it provides several more abstract functionalities regarding positioning, e.g.: `move_to(x,y)`, `move_relative(x,y)`, `moved_since?(time)` etc.

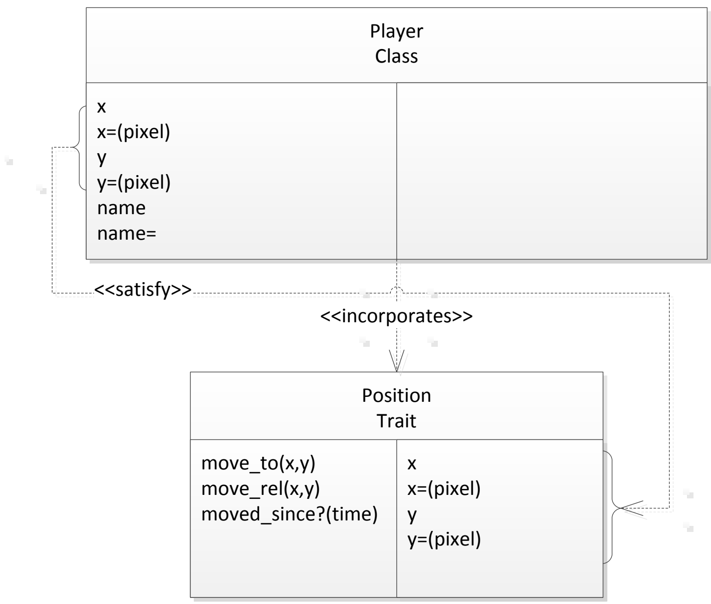


Fig 4: The graphical presentation of the example for provided, required and satisfied methods

The above diagram illustrates this example. At the top of each box there is the name and type of the entity. To the left of each box there are provided methods and to the right there are required methods. Classes may not have required methods as they are complete.

2.2.3 Symmetric composition

One important characteristic of a trait is its ability to be composed of other traits. This can happen in a hierarchical way or even in unstructured patterns. Basically any trait can incorporate any other trait. (cf. (Schärli, et al., 2003 p. 10))

Composition of traits means that the required methods of a trait can be partially or completely satisfied by the provided methods of its sub traits. Of course the composed trait can still be incomplete as its required methods may not be completely satisfied by its sub traits. Also the unsatisfied required methods of the sub traits become required methods of the composed traits. (cf. (Schärli, et al., 2003 p. 10))

Traits do not only have the characteristic of composition, the composition also needs to be symmetric. This means that the order in which traits are incorporated into another or a class does not make an impact on the resulting trait or class. (cf. (Schärli, et al., 2003 p. 16))

Example for composition.

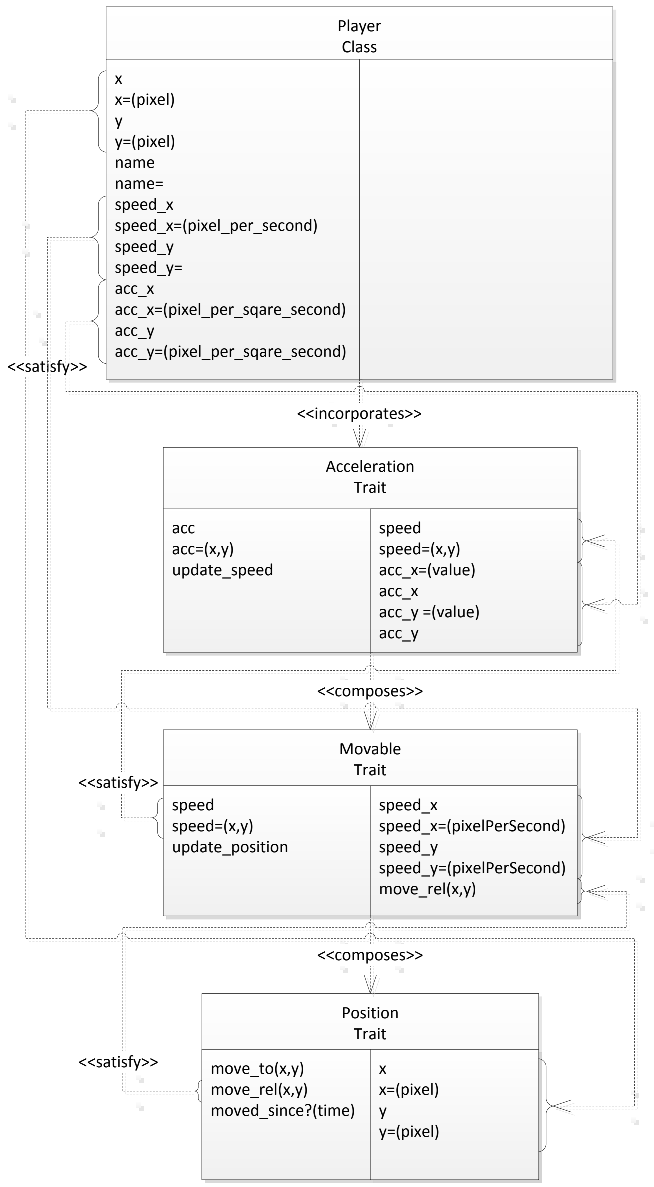


Fig 5: Diagram of example for composition

This diagram shows one class `Player`, that incorporates the trait `Acceleration`. Acceleration, however, is not a plain trait but a composed trait. The class Player does not know that Acceleration is composed, as it behaves as if it would implement and require all the methods from its sub traits. This transparence is required by the composition pattern. (cf. (Olsen, 2008 S. 111-125))

The same kind of transparence can be seen between `Player` and `Acceleration`. A user of `Player` cannot see what traits are incorporated - the class behaves just the same

as if it would implement all the offered methods from `Acceleration` and its sub traits by itself.
Reviewing this similarity among the relations of traits and classes, the <<composes>> labeling could be replaced by <<incorporates>>, as the same thing happens. The incorporator needs to satisfy the required methods of the trait, therefore getting that trait's methods provided. The unsatisfied methods become the required methods of the incorporator. The only difference between a class as incorporator and a trait as incorporator is the fact, that the resulting incorporator must be complete if it is a class. A trait as incorporator may not satisfy all required methods of its direct and indirect sub traits, therefore staying incomplete.

2.2.4 Explicit name conflict solving

As with every name based reference, there can appear name conflicts. This can happen, when there are two methods in two traits that have the same name. Even though they have the same name, they can mean totally different things. For example the method `moved_since?(time)` in `Movable` returns true if the position of an object on the screen has been changed since the given time. Another method with the same name `moved_since?(time)` in a trait `Resident` could return true if a person has changed his address since the given time.
When such name conflicts appear with traits, they need to be solved explicitly. This is one requirement for traits. (cf. (Schärli, et al., 2003 p. 11)) As the conflict solving is explicit, it can combine the different implementations in an explicit order, what means the order of conflict solving is independent from the order of incorporation. This implication of explicit conflict solving makes it easier to implement the symmetry of composition.

2.2.5 The composing entity offers the glue code

The requirement of explicit name conflict solving leads to the question where the code, that solves the conflict, shall be placed. What part of software is responsible for solving the conflicts? The trait definition in (Schärli, et al., 2003) answers this question as well: The incorporator needs to offer the so called glue code. This leads to two characteristics that are good in concerns of composition:

- The trait itself does not need to know with what other traits it is combined. The incorporator takes care of possible name conflicts.
- Composite traits solve their own inner name conflicts, so from an outside point of view, they are conflict-free and just as easy to use as non-composite traits.

2.2.6 Flattening traits into a class

A class that incorporates traits needs to behave as if it implemented the unconflicted methods itself via copy and paste. This holds especially for the classes' behavior within the class hierarchy. However, "Methods defined in a class itself take precedence over methods provided by a trait." (Ducasse, et al., 2006)

In (Schärli, et al., 2003) there is a tool presented, that can display smalltalk classes which incorporate traits, as a flat collection of methods. It is argued this helps understanding and maintaining classes that are built from traits. Such a tool is not subject to this work.

2.2.7 Recap: Characteristics of traits

In the previous chapters, six characteristics of traits where defined. In the following chart, these characteristics are condensed.

Table 2: Condensed characteristics of traits

Characteristic	Elucidation
Pure methods	Traits may only contain methods that do not access state directly.
Incompletion	Traits offer methods to the incorporator but may also require the incorporator to implement methods. These methods are called required. They are satisfied if the incorporator does indeed implement them and unsatisfied if they are not implemented.
Symmetric Composition	The order, in which traits are incorporated into a class, does not affect its consulting behavior.
Explicit conflict solving	Conflicts among different implementations of the same method must be solved explicitly.
Glue code	The code for explicit resolves is offered by the entity that incorporates the conflicted traits, not by the traits themselves.
Flattening into class	Methods offered by traits can be seen as if they were implemented in the class itself via copy and paste.

It is seen that by solving conflicts explicitly and therefore in an explicit order, the order of the incorporation of traits does not affect the behavior but the glue code does. Also, since the glue code is only offered in the class that caused the conflict, the traits stay untouched and can be reused without further consideration. It can be seen that the combination of glue code and explicit conflict solving leads to symmetric composition.

2.3 Traits, Mixins, Aspects

The above shown definition of a trait is designed to solve the problems with the different kinds of inheritance. It is of no surprise that there are also other approaches to solve these problems, two of which are selected to be circumscribed from traits.[6] Namely they are mixins and aspects. Mixins are already described in chapter 2.1.5 "Mixin inheritance" and traits are described in chapter 2.2 "A definition of traits".

Aspects are designed to solve so called cross-cutting concerns. These are concerns that are orthogonal to the class hierarchy, which means they don't interfere with the class hierarchy in a conceptual way. Examples for cross-cutting concerns are logging and authentication. The idea behind aspects is that these concerns should not bother the business logic as they are rather technical. This is another more detailed view on the separation of concerns.

The mechanic behind Aspects is to write the code that implements the cross-cutting concern – this code is called advice - separately from the code where it is applied to. Then this implementation is intertwined into the application code with join points. These can be defined explicitly or with a pattern like a regular expression. For example the aspect logging can be intertwined into all methods within the module Persistence that begin with "create". An aspect is the combination of join point and advice.

Aspects are most commonly defined by the two characteristics of obliviousness and quantification (cf. (Feigenspan, 2008 S. 4)) even though there is some discussion about whether obliviousness really is an immanent characteristic (cf. (Filman, 2001)).

Obliviousness means that the advice and join points are defined apart from the core business logic. A class `BankAccount`, for example, would be totally untouched from the aspects of logging and authentication. Quantification means that the join points can be defined at any location of the code, even within methods.

In practice, aspects are often used with join points within methods. Aspects enrich single methods or via a pattern a set of methods with functionality that cannot be seen in the definition of the method itself. This is vastly different from mixins and taits, where a method itself stays untouched, rather a class is enriched with new methods than a method with new aspects. For mixins and traits, a method can be seen as the atomic element of abstraction, while for aspects it is a line of code.

A second difference lies in the requirement of obliviousness. As defined in chapter 2.2.5 "The composing entity offers the glue code" incorporator of a trait has to offer the glue code, which results in at least one line of code even if there are no naming conflicts. The same goes for mixins - even though there is no explicit conflict solving for mixins, the incorporator still needs to declare, that it mixes in the mixin.

In the matter of conflict solving, traits only accept explicit solutions, whereas mixins only accept implicit solutions. In the second case, the programmer can only influence the conflict solving by changing the order in which the mixins are mixed in. For aspects, there is

[6] There are also other approaches to solve these problems like the strict prohibition of the diamond problem. This approach is described in (Malayeri, 2008) It is clearly different from traits and therefore not mentioned in this distinction.

no broadly accepted conflict solving strategy. A classification of behavioral conflicts between aspects as well as approaches for solutions is given in (Durr, et al., 2007). Also there is the approach to make composition or weaving symmetrical (cf. (Wunderlich, 2005)).

Table 3: Comparison of traits, mixins and aspects

Comparison of traits, mixins and aspects			
	Trait	Mixin	Aspect
Purpose	Building block for classes	Extending classes with disjoint functionalities	Extend a set of methods with non-business functionalities
Atomic element of abstraction	Method	Method	Line of code
Invasiveness	At least one line of code	Typically one line of code	Total obliviousness - 0 lines of code
Conflict solving	Explicit	Implicit	Not homogenous
Composition	Yes, symmetric	Yes, only linearly through single inheritance	Not homogenous, mostly asymmetric and implicit

The relation between traits and mixins is a very close one, since there are only a few abbreviations. However, functionality that can be achieved with mixins can also be achieved with traits but not the other way round. Therefore, the functionality of mixins can be seen as a subset of the functionality of traits – or speaking of classes, mixin could be a sub class of Trait. This relation also has influence on the hypothetical class diagram discussed in chapter 4.5 “Alternative implementations with more than plain ruby syntax”.

2.4 Known implementation of traits

To the knowledge of the author, there are several implementations of traits, two of which are for ruby. There is the gem “traited”, but instead of implementing the trait like it is defined in this work, it is “… allowing you to create a configuration similar to using class variables” (Jarvis, 2012). The other gem “traits” also manages the state of an object, especially its initialization.[7]

[7] “Traits” is documented at http://rubydoc.info/gems/traits/0.10.0/frames. It is developed by Ara T. Howard

For JavaScript, there is the extension traits.js[8], PHP supports traits out of the box as of version 5.4.0[9]. Moose[10] is an extension for Perl 5 that implements traits, while Perl 6 has "roles" built-in[11], which are traits with another name. Smalltalk provides traits in the two dialects squeak[12] and pharo[13]. Fortress[14] also takes advantage of the trait model.
One of the newest members in the "trait-family" is Scala, which has an artifact that is called trait. However, a closer look to scala's traits reveal, they are not actually traits in the definition of this paper, as they are described as following in the scala language specification v2.9 (Odersky S. 182):

- "traits are now allowed to have mutable fields."
  What offends the "pure methods" characteristic of traits.
- "…it is now possible to have overloaded variants of the same method in a subclass and in a superclass, or in several different mixins."
  What clearly breaches the "explicit name conflict solving" characteristic of traits.
- The class linearization (Odersky S. 64) indicates that the lastly mixed in "trait" overwrites behavior of earlier mixed in "traits", which is the behavior of a mixin.

Another not-quite-member of the "trait-family" is C++, which – referring to (Wikipedia, 2012) – has traits. However in (Alexandrescu, 2000) it is shown that "Traits [in C++] rely on explicit template specialization to pull out type-related variations from code, and to wrap them under a uniform interface." and therefore are rather a tool for enhanced generic programming but for composing a classes functionalities from reusable chunks of code.

[8] http://soft.vub.ac.be/~tvcutsem/traitsjs/, 10.3.2013
[9] http://www.php.net/manual/en/language.oop5.traits.php, 10.3.2013
[10] https://metacpan.org/module/Moose::Manual::Roles, 10.3.2013
[11] http://en.wikipedia.org/wiki/Perl_6#Roles, 10.3.2013
[12] (Schärli, et al., 2003 p. 17)
[13] http://www.pharo-project.org/home, 10.3.2013
[14] http://www.cs.cmu.edu/~aldrich/FOOL/FOOLWOOD07/Allen-slides.pdf, 10.3.2013

3 Ruby

In chapter 3, there will be given an introduction of ruby. The author assumes the reader knows about basic object oriented models as well as ruby syntax. Only those concepts of ruby that were used to implement traits will be shown.

3.1 Object model

In ruby, everything is a first class object. Instances of application-specific classes, strings, numbers, booleans and even classes, modules and methods are first class objects[15]. This can be illustrated with a diagram of an extract from the ruby standard library classes:

[15] The term first class object in this case means that an object can be stored in variables and data structures, can be passed as a parameter to a method, can be returned by a method and can be constructed at run-time. (cf. (Wikipedia, 2012))

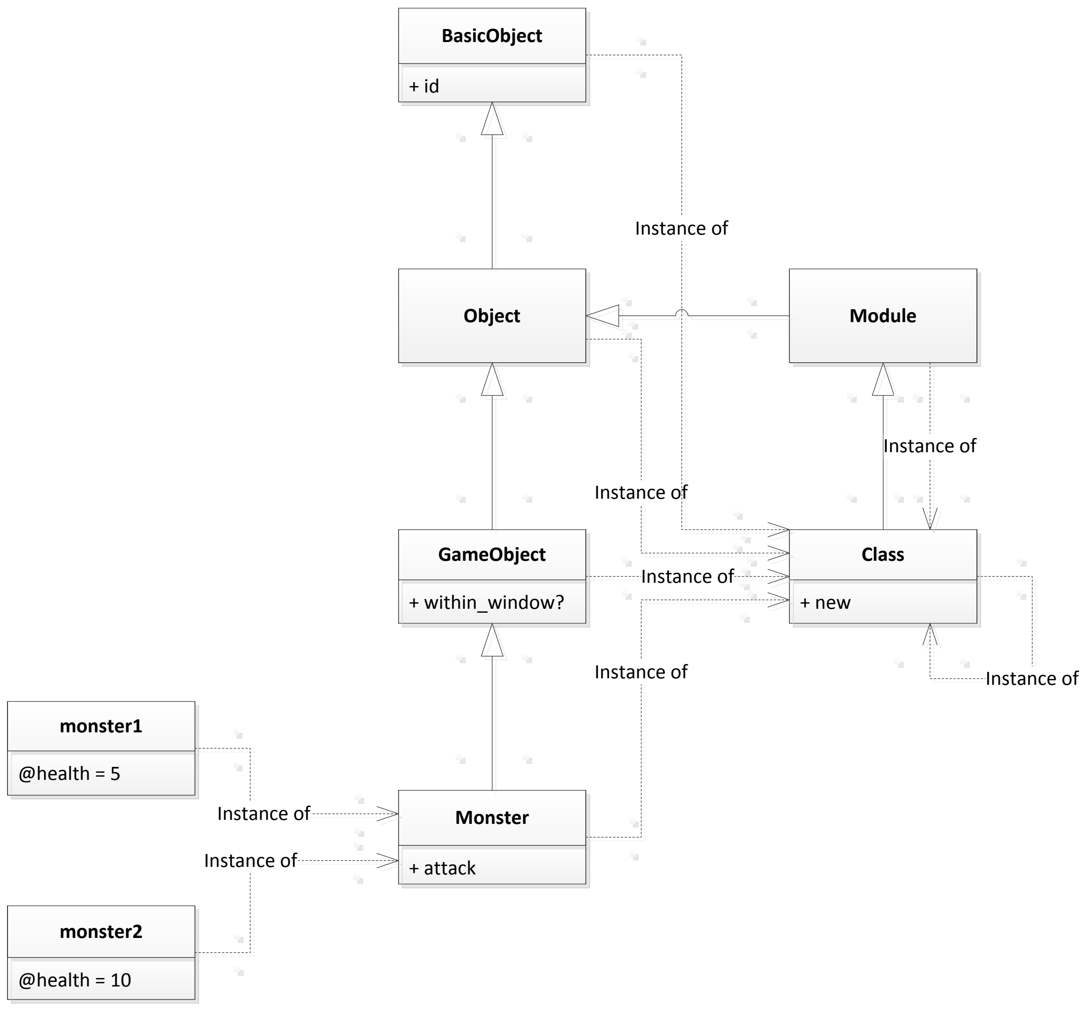


Fig 6: Example of ruby class hierarchy without Mixins. Adaption from (Perrotta, 2010 S. 23)

Everything that begins with a capital letter is a class, as can be seen with the "instance of" relation to the class `Class`. The UML-notation for generalization is a simple single-inheritance relation, read: "`GameObject` is the super class of `Monster`". Everything that begins with a small letter is an instance of a class that is not `Class`. In this case there are two instances of `Monster`, named `monster1` and `monster2`, that have one instance variable `health` with a number as value.
Something special is the relation between `Class` and `Module`. `Class` is a sub class of `Module`. So to say "`Class` is a special `Module`". And this is the case, really. A class is nothing else but a module, which can create instances of itself with the `new` method. While a module is just a collection of methods with a name, a class is a collection of methods with a name, which can create instances with an "instance of" relation to itself. This "instance of" relation will come into place with the description of method look up.

A module is a collection of methods - this seems familiar. It seems only natural that modules can be used as mixin, in ruby.
Also interesting is the “instance of” relation of `Class` to itself. Of course the class `Class` is a class, so it needs to be an instance of `Class`. This can be illustrated with the `new` method. Usually a new class is defined using the `class` keyword. But as classes are instances of `Class`, they can be created just like any instance of any class with the `new` method: `Class.new` creates a new class.[16]
The class `Class` defines the behavior of all classes in ruby. A method `new_valid_object`, that creates a new object via `new` and then performs a validation_check on that object, for example, would add this `new_valid_object` method to each class, even application-specific classes like `Monster`, `PowerUp`, etc.

3.2 Open classes

This idea leads to the concept of open classes (cf. (Perotta, 2010)), as it is actually possible to add such a method to `Class`, thus defining a new way of object creation for all classes. A class (and module) can be reopened at any time, adding new method or redefining existing ones, using the same `class` (or `module`) keyword, that is used to define a totally new class. In practice, this means that class definitions can be spread over multiple files, as different responsibilities of a class can be defined at different places.
The concept of open classes holds for application-specific classes as well as classes from the ruby standard library.

3.3 Method look up

When a message is sent to an object, it consists of three parts: The receiver object, the name of the method and the parameters. When the receiver object receives the message, it has to find the definition of the method with the name sent within the message to execute the code under the definition within the context of the given parameters. This process is called method look up. In ruby, it depends on the “instance of” relation of the object to its creator class and the “super class” relation of that creator class. According to the way of illustration that is used in Fig 6 the method look up is called “one step to the right, then up”. At first, the receiver object looks for a method definition in its creator class. When there is no definition with the given name, it continues its look up at the super class and the super class of the super class etc. until there is no super class (`BasicObject`). (cf. (Perrotta, 2010 S. 25))
The order in which the classes are addressed during the method look up is called ancestors. The ancestors are also first class objects and can be accessed via the `ancestors` method

[16] This class would not be bound to a constant (its name) with this code alone, thus leaving it an anonymous class. This can be fixed by assigning it to a constant using the method `const_set` in the class Object. More details can be comprehended in “Metaprogramming Ruby” by Paolo Perrotta (cf. (Perotta, 2010)).

of the creator class of the receiver object. The ancestors of `monster1` would be [`Monster`,`GameObject`,`Object`,`BasicObject`]. The ancestors of `Monster` (the method is sent to the class `Monster` itself, not to an instance of it) would be [`Class`,`Module`,`Object`,`BasicObject`].[17]

3.4 Eigenclasses

Eigenclasses are also called meta classes or singleton classes. The phrase "eigen" refers to the german word *eigen* that means "something's own". It belongs to only a single object and each object has its own eigenclass. Within the class hierarchy it is placed directly between the object and its creator class (cf. (Perrotta, 2010 S. 116)). An example for this is illustrated in the following diagram:

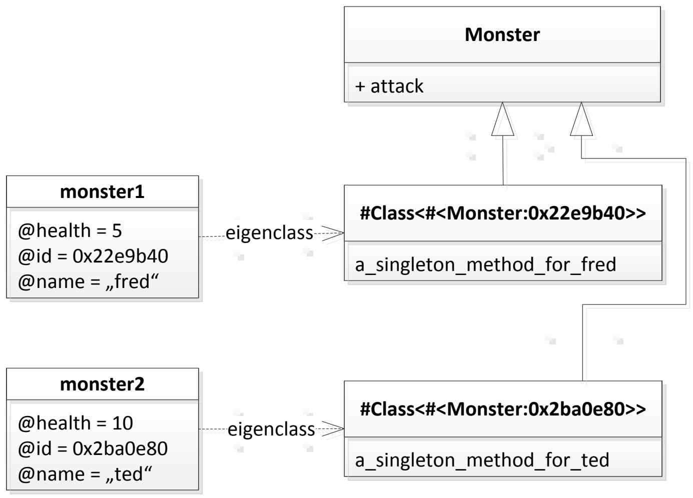


Fig 7: Eigenclasses of two objects

What does this mean for the method look up? It works with the same pattern as before: When a message is sent to an object, the method look goes one step to the right and then up. So at first the eigenclass of the object is searched for the method, then all the super classes.

There are several practices that make use of eigenclasses in ruby. For example class methods are just methods in the eigenclass of a class. As seen in chapter 3.1 "Object model" classes are first class objects – plain instances of the class `Class`. Messages cannot only be sent to the instances of classes but also to the classes themselves, these methods are called class methods. In chapter 3.3 "Method look up" there is illustrated that methods are only defined in classes while instance variables always belongs to the instance. So if a

[17] These ancestors are not the actual ancestors that would be returned by a real ruby program, as in this demonstration mixin modules and eigenclasses are left out due to simplicity.

method is sent to a certain instance of class and the message is understood (in other words: The Class instance can execute the method) the method would be defined for all classes, as all instances of a class respond to the same set of messages. This would only be the case if there were no eigenclasses. With eigenclasses, each instance of class can have its own set of messages that it can respond to. There are several syntaxes to define a method for an eigenclass.

```
 1 class Monster < Struct.new(:Monster, :name, :health)
 2   # Class method Syntax 1:
 3   class << self
 4     @@numbers = 0
 5 
 6     def number_of_monsters
 7       @@numbers
 8     end
 9 
10     def new *args
11       @@numbers += 1
12       super
13     end
14   end
15 end
```

Fig 8: Defining a class method with the class << self syntax

With `class << self` the context of the eigenclass of the current `self` can be entered. `Self` in this case refers to the class `Monster`. Thus everything between line 3 and 14 is executed in the context of `Monster`'s eigenclass. The class variable `@@numbers` as well as the two methods `number_of_monster` and `new` are defined for the eigenclass. The keyword `super` will execute the `new` method of the eigenclasses super class, which is `Monster`.

```
1 class Monster
2   # Class method Syntax 2:
3   def self.number_of_monsters_twice
4     @@numbers * 2
5   end
6 end
```

Fig 9: Defining a class method with the def self.method syntax

Instead of entering the context of the eigenclass and then defining the method, there is a shortcut to define a method for a single object. The `def` keyword is not only used with the name of the method but also with the object that the method is defined for. In this case the current `self` – `Monster` – is that object. The method `number_of_monsters_twice` is defined only for this single object that is an instance of Class.

This shortcut with the keyword `def` cannot only be used within classes it can be used with any object reference. The same goes for the `class << object` syntax.

3.5 Mixin modules

A mixin module, in ruby, is an instance of `Module` that is used as a mixin. Mixin are described in chapter 2.1.5 "Mixin inheritance". As mentioned in chapter 3.1 "Object model", a module is a named collection of methods. The methods do not need to be pure, meaning they are allowed to access state directly.

How does a regular module become a mixin module? It is simply included by a class or another module using the `Module#include`[18] method. An example of how mixin modules are defined and included is given in the appendix 11.1 "Code example for composition of mixin modules". The inclusion of a mixin in ruby works in the way mentioned in chapter 2.1.5 "Mixin inheritance". A pseudo-class with the methods of the module is inserted directly above the including class in the class hierarchy (cf. (Perrotta, 2010 S. 26)).

Thus the class itself and all its sub classes have access to those methods but not its super classes. It seems as if the methods where defined in the class itself – only they can be reused at another place in the class hierarchy seamlessly.

As displayed in the example, there can be conflicted methods that have the same name - in this case `update`. Ordinarily the version of `update` that was included as last is the first in the ancestors and therefore the first one that the method look up will find. It is the only version that is being executed when the method `update` is sent to the object. However, this problem can be fixed using the keyword `super`, which continues the method lookup from where it stopped.

The disadvantages of linear asymmetric composition are discussed in chapter 2.1.5 "Mixin inheritance".

It is noted, that in ruby, there is the possibility of something such as composition of mixins. This can be done with the hook method `included` that is called on a mixin module upon inclusion. In this method it can make the including class including further mixin modules. The including class explicitly only includes one mixin module, but rather includes other modules as well. Those other modules may as well include further modules etc. This can be used to automatically include required mixins, on which the included mixin is based on. The including class of the included mixin does not notice it actually includes several other mixins, due to that mixin in turn including several other mixins. This behavior is very composition-like. However, the class may still include another mixin, that has methods with the same names. Then it once again comes down to the order in which the mixins were included.

[18] The notation `Module#include` means: „The instance method `include` that is defined in the class `Module`".

3.6 Class and module Macros

In ruby, a macro is an ordinary method that is sent to a class or a macro. It changes the behavior of the receiver and is often used in a rather declarative way. The most popular example of a class macro is the method `attr_accessor` that defines instance variables for a class.

3.7 Reflections

As many languages, ruby enables a programming technique called reflection. Basically it describes the capability of a program to analyze its own structure as well as behavior.
Although there is a wide range of reflection techniques in ruby in this chapter there will be displayed only two of them: Hook methods and the `methods` method family.

3.7.1 Hook methods

There are two concepts that refer to the phrase "hook method". One of them has the meaning of a method that implements a default behavior in a common super class. This behavior is meant to be overwritten in some sub classes. An example for this is an attribute method `text?` that is implemented in Object and returns false. All classes that represent a text, like String or Symbol, would overwrite the method and return true. Therefore all the other classes do not need to implement the `text?` method. This concept of hook methods is not a kind of reflection and is not the concept that is used in this work. In this work, the phrase hook method, refers to a method that is empty at default only to be filled by the application programmer. Hook methods are placed at certain points in the program to give an opportunity to catch certain events. One example for this kind of hook method is `Class#extended`. It is called whenever a class gets a new sub class with that sub class as the parameter. Hook methods can be considered a reflection technique as they do not give information on the structure of the program, but on the behavior of the program.
In chapter 3.5 "Mixin modules" the hook method `Module#included` was already mentioned.

3.7.2 Methods

What methods does this object respond to? How many arguments does this method accept? Is this method private or public? Is this method defined in the objects generator class or in a super class?
All those questions are not only important to a programmer. When it comes down to manipulating a class or object at runtime, this information must be accessible not only via the documentation or reading the source code but also programmatically. In ruby, every object responds to methods regarding this information. The most common and most general example is the `methods` method. It returns the names of all methods that are defined for the object.

Example.

```
"A string".methods.inspect #  =>  [:<=>, :==, :===, :eql?, :hash, :casecmp,
                           #:+, :*, :%, :[], :[]=, :insert, :length, :size,
                           #:bytesize, :empty?, ...]
```

Fig 10: The methods method returns an array of method names. This is only an excerpt; all methods can be seen in appendix 11.2

Other `methods` methods are: `singleton_methods`, `protected_methods`, `private_methods` and `public_methods`.[19] Further information about a method can be gained by using the `method` method that takes the name of the method as parameter and returns an instance of the class `Method`.
Example.

```
"A string".method(:to_s).arity # => 0
# The method "to_s" for the object "A string" does
# not take any parameters

5.method(:class).owner # => Kernel
# The method "class" for the object 5 is defined
# in the module Kernel
```

Fig 11: Some methods of the class Method

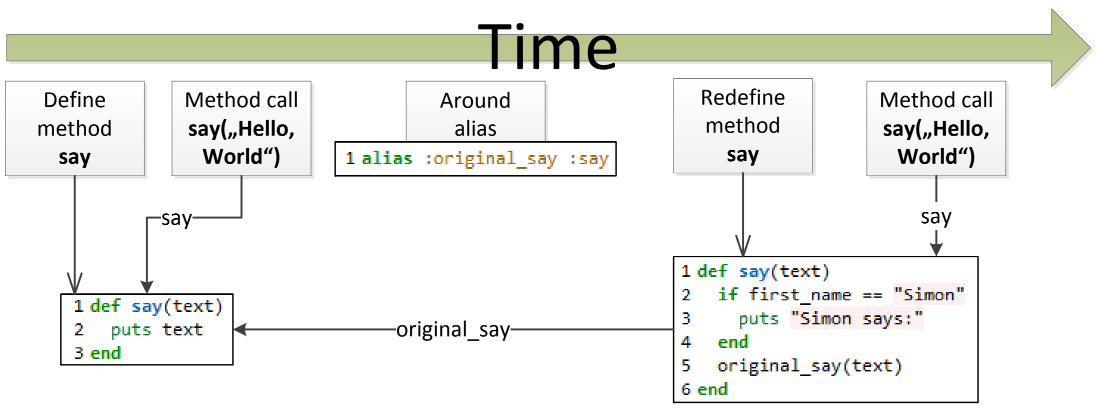
Another technique for reflecting about methods is the method `respond_to?`. It has one parameter that is the name of the method, the object returns true or false without executing the actual method. `Respond_to?` can be used to check whether a method can be understood by an object before sending the actual method. In the latter case a singleton method or even a regular method might be defined for this object before the actual message is sent.[20]

3.8 Aliasing

The trait model requires conflicted methods to be able to be renamed, so they can be accessed independently from the order in the method look up. Ruby has such a mechanism called aliasing. A method can be aliased using the `alias` key word. It then is available under both names. Even when the original method implementation is redefined, the alias still is linked with the implementation that was present at the time of aliasing. This behavior can be used to add new functionality to an existing method without having to alter the class hierarchy. This is called **around alias** (cf. (Perotta, 2010 S. 132)).

[19] The concept of singleton classes and singleton methods are not further regarded in this work, as they are particular important to the implementation of traits.

[20] A method that is generated at runtime is called Dynamic Method. (cf. (Perrotta, 2010 S. 44,45))



```
def say(text)
  puts text
end
```

```
def say(text)
  if first_name == "Simon"
    puts "Simon says:"
  end
  original_say(text)
end
```

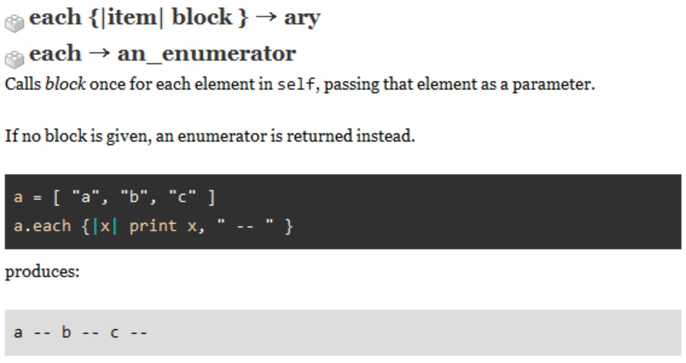
Fig 12: In this around alias the old `say` method just puts out its text parameter. Then the original `say` method is aliased with `original_say`, it is still available under that name even when in the next step the method `say` is redefined. Within the new definition of `say`, there is a message sent named `original_say`, which is the `say` method from before the around alias. So the resulting new `say` method contains behavior of its prior version.

An around alias involves the following steps: (cf. (Perotta, 2010 S. 132))

- Alias the original method with a unique name
- Redefine the method with the new functionality
- Call the original version using the unique name from step 1

3.9 Blocks, Lambdas and Procs

The lambda calculus was introduced by Alonzo Church in the 1930s to formalize the concept of effective computability. Up to now it has provided a strong theoretical foundation for the family of functional programming languages (cf. (Rojas, 1998)).
In ruby, lambdas are dynamically scoped functions which can be used in many different ways; however the iterator application of lambdas or blocks is the most common: The method `each`.



each {|item| block } → ary
each → an_enumerator

Calls *block* once for each element in `self`, passing that element as a parameter.

If no block is given, an enumerator is returned instead.

```
a = [ "a", "b", "c" ]
a.each {|x| print x, " -- " }
```

produces:

```
a -- b -- c --
```

Fig 13: The official documentation for Array#each

In Ruby, there are three different implementations of lambdas: `Blocks`, `Lambdas` and `Procs`. While `Lambdas` and `Procs` are first class objects, `Blocks` are not. Often, it is not needed to have an object reference to the block as it is only called where it was defined. For this case, there are `Blocks` which can be executed via the `yield` method. A Block can be turned into a `Proc` using the `&`-Notation.

```
def my_method(parameter1, &block)
  puts block.inspect #block is a Proc instance
end

my_method("something") {"block returns this string"}
#=> #<Proc:0x289b378@C:/path/to/source/file.rb:5>
```

Fig 14: Transforming a block into a Proc instance using the &-syntax

Procs and Lambdas are very similar as they are both first class objects and even respond to the same set of messages. Both are instances of the same class Proc but Lambdas are flagged as such. This leads to different behavior regarding parameters and the keyword return: Lambdas are more strictly regarding missing or abundant parameters while keyword `return` aborts only the execution of the lambda itself. In a Proc, `return` aborts the execution of the caller.[21]

```
def return_in_lambda
  (lambda { return true }).call
  false
end

def return_in_proc
  (Proc.new { return true }).call
  false
end

return_in_lambda     # => returns false
return_in_proc       # => returns true
```

Fig 15: The differences of the keyword super for Lambdas and Procs

So there are three kinds of implementations for the lambda calculus in ruby: blocks, lambdas and procs. The difference can be comprehended in the following table 4.

[21] The exact differences between procs and lambas and ways to create both are comprehended in the official ruby documentation (cf. (ruby-doc.org)).

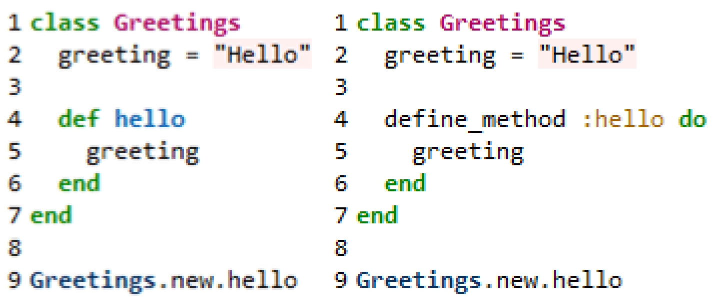
Table 4: Comparison of blicks, lambdas and procs

Comparison of blocks, lambdas and procs			
	Block	Lambda	Proc
Return	Not allowed	Aborts execution of lambda	Aborts execution of calling method
Parameter	Loose	Loose	Strict
Instance of	Not first class object	Proc	Proc

There are three implementations for only one concept of the lambda calculus. This is a valid approach as there are several possibilities for improving performance for the different kinds of uses of lambdas. Lambdas without parameters might be implemented differently from others. Or lambdas that do not need to access the calling stack because they only use constants and internal local variables might once again be optimized in another way. However, the author disapproves that the different kinds of optimizations are visible to the application programmer. These kinds of optimization can and should be done on compiler level, as three implementations for only one concept discount the principle of least surprise and only lead to confusion.[22]

So, Lambdas are anonymous functions that are scoped dynamically. Dynamic scope means that "at any point in time during the execution of a program, its [the functions] binding is looked up in the current call stack as opposed to the lexically apparent binding as seen in the source code of that program." (Costanza, 2003 S. 1)
However, in ruby the bindings are not only looked up in the current call stack, but – as a backup – in the lexical surroundings of that block, as well. This behavior can be comprehended considering this example.

[22] The confusion that is caused by the three different implementations can be comprehended among others in those two discussions: http://stackoverflow.com/questions/2983907/im-confused-with-block-in-ruby-compared-to-smalltalk, http://stackoverflow.com/questions/1386276/difference-between-block-and-block-in-ruby?rq=1. Language design, however, while being an interesting field of discussion is not subject to this work. Therefore the discussion about lambdas is not spread any further.



```
1 class Greetings
2   greeting = "Hello"
3
4   def hello
5     greeting
6   end
7 end
8
9 Greetings.new.hello
```

```
1 class Greetings
2   greeting = "Hello"
3
4   define_method :hello do
5     greeting
6   end
7 end
8
9 Greetings.new.hello
```

Fig 16: Dynamical scope of blocks

In the left example, the method `hello` is defined with the keyword def. In line 9 there will be thrown a `NameError` as the term `greeting` in line 5 does not refer to the variable `greeting` in line 2. This local variable (line 2) is not in the scope within the method (lines 4 – 6).
In the right example, the method `hello` is defined using the `define_method` method that uses a block to specify the code that is executed upon calling the defined method. Within this block (lines 4-6), the local variable `greeting` from line 2 is visible and the call in line 9 will return "`Hello`".[23]
This means that a variable that is within class scope – in this case `greeting` – is usually not visible within methods in that class. Similarly a variable that is within the scope of a module is not visible in a class that is defined within that module. Modules, class methods are so called scope gates as they open a new scope. However, this can be circumvented with using blocks to define modules, classes or methods. Blocks also have their own scope but in addition they can refer to the parent scope, as illustrated in the example from Fig 16. This behavior of blocks is referred to as **flattening the scope** (cf. (Perrotta, 2010 S. 79-81)).

3.10 Named Parameters

Ruby has no real named parameters. There is, however, a compensation for that. It is basically syntax for creating hashes that can only be used within the parameters of a method call. So in fact, the method only has one parameter that is a `Hash` but the method call looks like there are many different parameters with names.[24]

[23] `greeting` is not a instance variable but a local variable in the scope of the class `Greeting`, which can be seen as a name space in this example.

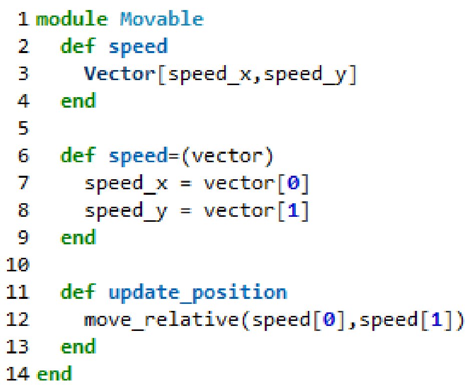
[24] It is noted that during the writing of this work the Ruby 2.0.0 was released. Ruby 2.0.0 has a new, more dedicated feature for named parameters. (cf. http://www.ruby-lang.org/en/news/2013/02/24/ruby-2-0-0-p0-is-released/, 10.3.2013)

4 Extending ruby with traits

As the practice part of this work an implementation of traits is proposed. In the following chapter it will be discussed how the techniques described in chapter 3 can be applied to implementing traits for ruby. At first an introduction to the implementation will be given in form of a case study of how traits are defined and how they are used to build classes with them.

4.1 Trait definition

As discussed in chapter 3.1 “Object model”, a module is a collection of methods that has a name. As a trait also is just a collection of methods, it seems only natural to use a module to define a trait in ruby.



```
 1 module Movable
 2   def speed
 3     Vector[speed_x,speed_y]
 4   end
 5
 6   def speed=(vector)
 7     speed_x = vector[0]
 8     speed_y = vector[1]
 9   end
10
11   def update_position
12     move_relative(speed[0],speed[1])
13   end
14 end
```

Fig 17: A definition of a trait in ruby

This is the definition of the trait `Movable` as it is used in Fig 5. It provides the methods `speed`, `speed=(vector)` and `update_position`. It is easy to see that provided methods are those that are defined in the module. The required methods are not as easy to identify as they are just used within the implementation. The required methods are `speed_x` and `speed_y` in line 3, `speed_x=(pixel_per_second)` in line 7, `speed_y=(pixel_per_second)` in line 8 and `move_relative(x_pixel,y_pixel)` in line 12.[25]

[25] `speed_x=(pixel_per_second)` and `speed_y=(pixel_per_second)` are clean methods as they are just setters for those state fields. These methods could encapsulate any form of representation of state.

As of now, there is no explicit detection for unsatisfied methods. Only a call on an unsatisfied method will give feedback to the programmer in the form of a `NameError`. This leads to a more lean way of defining traits as they are seemingly not different from mixins. Any form if explicit enumeration of required methods would lead to more effort in implementation as well as redundancy. The listing of required methods is left to documentation for those reasons.

4.2 Building classes with traits

Defining traits is only one half of the work for the application programmer as traits are used to build classes from them. Fig 18 shows an example of a class flat hierarchy with making use of traits. The n to m relation among classes and traits is illustrated.

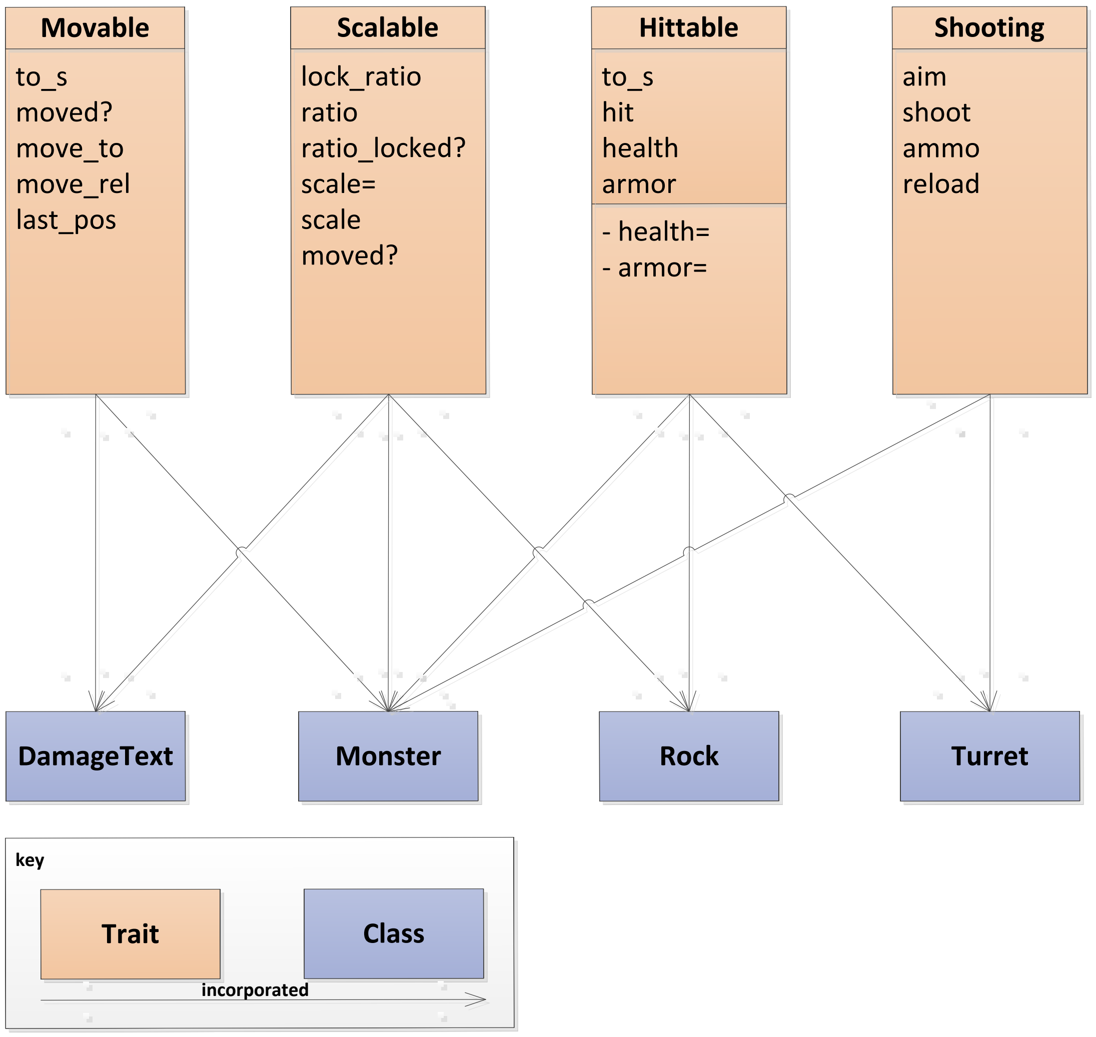


Fig 18: 4 classes are built from 4 traits.

The actual syntax for incorporating traits into a class is available in two forms: As a class macro[26] or with a builder[27]. Fig 19 shows how the `Monster` class from Fig 18 is built with traits.

```
 1 class Monster
 2   include Traitable
 3   incorporate.
 4     trait(:movable).
 5       with_options(except : :to_s).
 6     and.
 7     trait(:scalable).
 8     and.
 9     traits(:hittable, :shooting).
10     resolves(:moved?).
11       with_lambda(-> { raise "Overwrite method 'Monster#moved?' to resolve" }).
12     resolves(:to_s).
13         with_pattern.
14         call_in_order.
15     done
16 end
```

Fig 19: The class `Monster` is composed of four traits: `Movable`, `Scalable`, `Hittable` and `Shooting`

In line 2, the mixin `Traitable` is mixed into the class. This provides the two class macros `trait` and `incorporate`, second of which uses implements pattern. Incorporate returns the builder object, which receives methods that define an incorporation and always return the builder object again. So with a long method chain of `trait`, `traits`, `resolves`, etc. an incorporation definition is declared. The `done` method in line 15 is called last to trigger the actual incorporation into the class.

Technically everything from line 3 to 15 is one statement with the class as receiver. The fact, that all traits and all resolves must be declared in one statement is a design decision that was made due to simplicity. When traits have conflicted method names, there must be a way to detect them all and resolve them in a symmetrical fashion. This would be much harder when there could be new traits incorporated at any time. With each new incorporation definition, there could emerge new conflicts which would need resolving. But the order in which they are resolved could influence the semantic of the resolve. So in the end the order in which traits would be incorporated could influence the behavior of the resulting class. This would be the opposite of symmetric composition.

In the following chapter there will be more design decisions spread out.

[26] A class macro is a method that is sent directly to a class or module, modifying the behavior of that class or module (cf. (Perrotta, 2010 S. 114,115)).

[27] The builder pattern is described in (Olsen, 2008 S. 249-260) among others.

4.3 Implementation of traits

The proposed implementation of traits is based heavily on mixins. Mixins already provide a lot of functionality which is needed for traits as well:

- They add methods to classes and
- They can be composed, though only linearly.

There are only a few things that distinguish mixins from traits:

- Explicit conflict solving,
- Real composition instead of linear composition and
- The possibility to include only a few methods of a trait.

So what the proposed implementation does in its core is to extend Rubies mixin model so that they

- Can be composed in any pattern, not just linearly,
- Solve name conflicts explicitly, not implicitly,
- Provide a way to use only few methods, not always all the methods.

The following table recaps what is accomplished with the proposed trait implementation.

Table 5: The core ideas of the proposed trait implementation

The core ideas of implementing traits via mixins		
Mixins (initial point)	**Traits (target point)**	**How it is accomplished (transition)**
Implicit conflict solving	Explicit conflict solving	Conflict detection via `methods` method; Aliasing and redefining original methods
Linear composition	Real composition	Explicit conflict solving, thus explicit ordering
Inclusion as a whole	Can be incorporated partly	Except and only filter

4.3.1 Conflict detection

A method name conflict emerges when there are two or more traits incorporated into a class that implement two or more methods with the same name. These conflicts are detected upon the actual process of incorporation.

The underlying data structure is an `Incorporation` that aggregates `Trait` instances within a hash. The traits are keys in the hash in which the values are options for that particular incorporation. Options again are hashes, with either `:except` or `:only` as the key of the `Hash` and an array of symbols as the value. These symbols represent methods that shall be filtered via `:except` or `:only`. This data structure already expresses the

symmetrical nature of trait incorporation as there is no particular order in the key-value-pairs within a hash.
To detect conflicting method names, a trait has a method `colliding_methods` which iterates over all trait-options-pairs in the traits hash. It accumulates all implemented methods of the traits under the given options[28] and also adds the instance methods of the incorporator, as method conflicts cannot only occur among traits but also between the incorporating class and one of its traits.
After all methods from all traits and the incorporator are accumulated, they are filtered for duplicates which are then returned in an `Array`.[29]

4.3.2 Conflict solving

Resolving a name conflict must be done explicitly and in a symmetric fashion in the composing class or trait. This is done with Lambda, which defines the new implementation of the conflicted method. Often, it is needed to access the original implementations of the traits. To be able to do so, they need to be aliased. So resolving name conflicts relies on two techniques: Aliasing different implementations and defining the resulting method.

Aliasing conflicted methods is trivial as the aliasing technique described in chapter 3.8 "Aliasing" applies to mixin modules. The problem is rather that a trait has no knowledge of where it is incorporated and whether it is being used for composition or not. So it does not know what methods need to be aliased. Therefore it provides a method `alias_methods(*method_names)` that receives a list of method names as parameter. It aliases the given methods so that the new name is the old name plus a suffix that depends on the traits name. The method `update` in the trait `Acceleration` would be aliased to `update_in_acceleration`. Which methods need to be aliased depends on the incorporation, as it detects the conflicts under the given combination of traits and options (cf. chapter 4.3.1 "Conflict detection"). However, this approach to aliasing is optimized in that regard that only conflicted methods will be aliased. Once a method is aliased, it is a firm element of that trait and will be incorporated into other classes as well. This is not too bad as they are unique and their expressive name may not lead to confusion. But each incorporation will detect the same conflict and alias the method again, although it has already been aliased in the first incorporation. This is not harmful as well, other than it is time inefficient and conceptually not clean. The author sees two ways of approaching this problem.

- Mark methods as aliased once they are aliased so they are not aliased again

[28] To find those methods of each trait, the trait itself has a method that filters its own provided methods with the given options.
[29] The `Array#duplicates!` and `Array#duplicates` methods are also part of the proposed implementation and take advantage of the open class principle described in chapter 3.2 "Open classes".

- Alias all methods blindly upon trait definition even before an incorporation is defined

The first solution arranges it so that only necessary aliasing occurs. It has an overhead in memory and in code, as another state for the trait object would need to be defined. In the proposed implementation traits are stateless, just like mixins.
The second solution leads to much less and easier to understand code. It has the downfall of probably aliasing too many methods and therefore polluting the resulting class with many methods that are not needed.

After aliasing the different implementations is done, the second technique of **redefining** the conflicted method comes into use. This is achieved via the `define_method` method, that is also used in chapter 3.9 “Blocks, Lambdas and Procs” to illustrate the dynamic scope of blocks. The `define_method` method defines a method, where the first parameter is the name of the new method and the block of the method is the code of the new method. The code in the block will be executed in instance context. Therefore instance methods – like the aliased conflicted methods – can be referenced simply by their name. An example for that is given in line 9 of appendix 11.3 “Building a class with the class macro “trait””. By redefining the conflicted method with an explicitly written Lambda there are several requirements of traits assured.

- Conflict solving is explicit.
  This is pushed even further by the explicit detection of unresolved conflicts during incorporation.
- The resolve for each method is symmetric, therefore the composition of traits are symmetric.
  The order and the context in which original implementations of the conflicted method are combined, is defined in a Lambda in explicit, sequential code. The place in which the Lambda occurs does not change the actual code within it, nor does the order in which traits are incorporated, as each conflicted method is aliased to a unique name that is then used in the Lambda.

In practice there occur often similar problems that can be solved in similar fashions. The incorporator may want to call all implementations of the conflicted method. Maybe he does not only want to call them but also combine them in a certain context, such as `+` for Strings. Or he may pick only one implementation. These similarly solutions to method conflicts are called **patterns**. The incorporation builder provides methods to implement these patterns for certain, conflicted methods. These are just shortcuts to reduce the effort of implementing simple resolves – especially when there are many traits combined. It does not change the underlying structure of traits, as all these patterns are eventually just Lambdas, which could have been hand written as well.

Recap: Conflict solving is done in two steps.

- Alias all conflicted methods
- Redefine the resulting method by calling all conflicted methods by its alias

4.3.3 How Traits are built from mixins

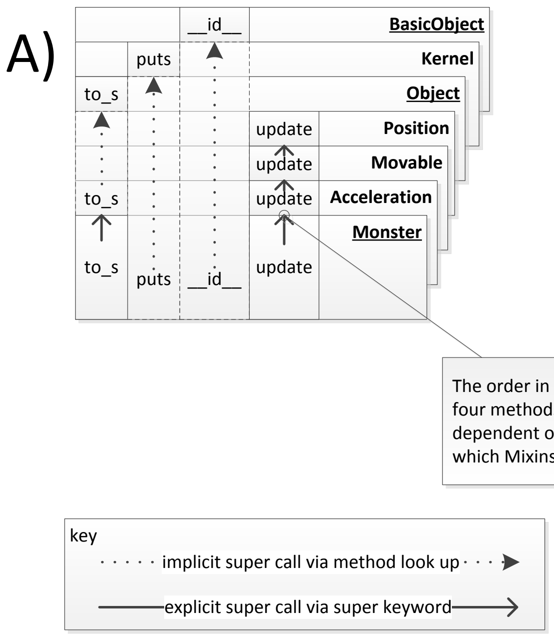

A)
B)
BasicObject
Kernel
Object
Position
Movable
Acceleration
Monster
__id__
puts
to_s
update
The order in which these four methods are called, is dependent on the order in which Mixins are included
key
implicit super call via method look up
explicit super call via super keyword

Fig 20: In part A) of the above figure, the `Monster` class with its three mixins `Acceleration`, `Movable` and `Position` is shown. Also its super class `Object` as well as `Objects` mixin `Kernel` and super class `BasicObject` are shown, thus all the ancestors of `Monster` are visible in this example. The tiering of those modules can be seen as a 3D view from slant above. Actually, the `Monster` class stands in front of the other modules and covers it. This means that the user of `Monster` can only see those methods that are written down at the bottom of Monster. All the other modules are only part of the implementation of `Monster`, not of its interface. It can be seen that an instance of `Monster` responds to the message to_s but the class `Monster` does not actually implement that method. It is only found in `Object` by the method look up. Part B) shows the exact same class only that the ordering of ist mixins is slightly altered. Acceleration and Movable are twisted compared to part A.

It can be seen, that due to the alternation in the ordering of the mixins and the linear nature of mixin composition, the ordering in which the different update methods are called, has also changed. In part A the ordering would be `Moster#update`, `Acceleration#update`, `Movable#update` and then `Position#update`. In Part B it is `Moster#update`, `Movable#update`, `Acceleration#update` and then `Position#update.`
Considering, that the calculation if the speed vector is dependent from the acceleration, it makes a difference whether the speed vector is increased after or before the acceleration

has been increased. The ordering of the mixins therefore heavily impacts the behavior of the `Monster` instance. The asymmetric nature of mixins becomes obvious.
The most important portion of Fig 20 is the comment box right in the middle. It stresses the fact that the programmer cannot influence the ordering in which the `update` versions are called once the ordering of mixins is set. Adjusting the ordering of mixins to fix the order in which methods are called, is no solution as it is possible that for `update` the ordering in part A is a good one but for another method the ordering in B might be the right one. The problems with this kind of composition are discussed in chapter 2.1.5 “Mixin inheritance”.

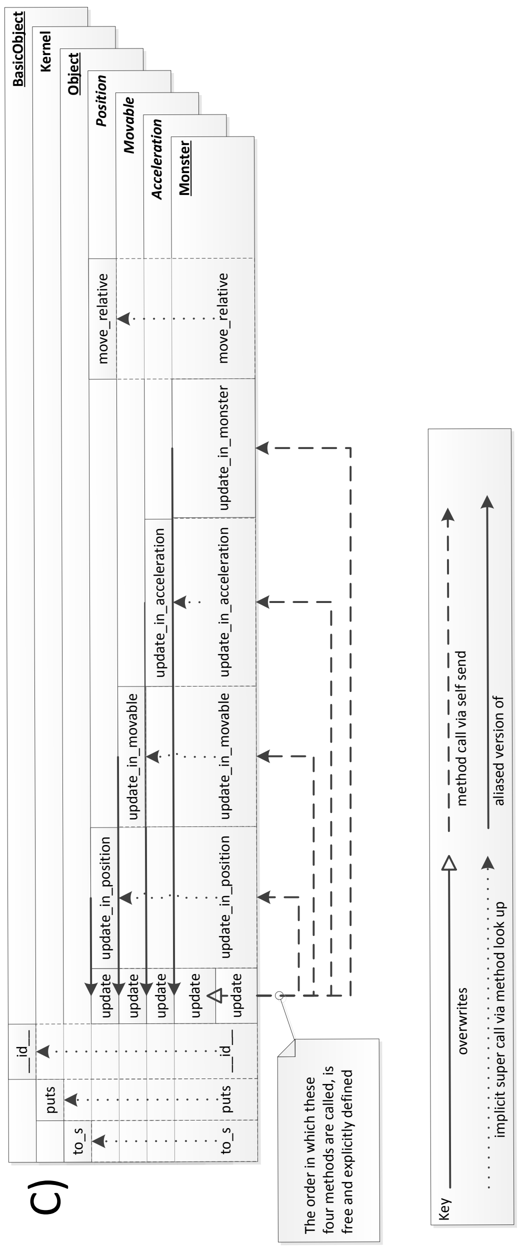


Fig 21: This figure uses the same syntax as Fig 20. Once again the `Monster` class is illustrated but rather than using mixins, traits are used to implement `Monster`. Therefore, `Kernel` is the only mixin and `Acceleration`, `Movable` and `Position` are traits (indicated with their italic font). A key difference between the two figures can be found in the key. In part C there is no explicit call of super. Instead there are a lot of aliased versions of the method `update`, one for each trait and class that implements it.

There is a subtle part in this figure that needs to be highlighted. In the class `Monster` there are two versions of `update,` both of which seem to be visible to the user. This is due to the lack of time in this illustration. In fact the two versions of `update` show an around alias as mentioned in chapter 3.8 “Aliasing”. After the around alias has been applied only the bottom version of update is known under this name while the upper version is known under the alias `update_in_monster` as indicated to the far right of Fig 21.
It is noted that there is no aliased version for `move_relative`, as it causes no conflict. `move_relative` is just found by the method look up. A more complex solution aspired with the method update, as it is defined in all three traits and the incorporating class. A four-way name conflict is present as the `update` in `Monster` covers up the `update` in `Acceleration`, `Movable` and `Position`. This is solved in two steps:

- The conflicted methods are **aliased** within the respective module[30] in a unique way. Now, the aliases of the conflicted methods are visible to the user of `Monster`, which means they can be accessed via the ordinary method look up without having to use the keyword `super`.
- The aliased methods are **combined** in an around alias. The around alias is introduced in chapter 3.8 “Aliasing”. A new `update` method in `Monster` is defined that overwrites the old `update` method, which is still available under its alias `update_in_monster`. Within the new `update` method all the other or only a few are called in an explicit order that is independent from the order in which `Acceleration`, `Movable` and `Position` where included.

Once again, the most important part if this figure can be found in its comment box. In contrast to Fig 20, the ordering in which the different versions of update are called, is independent from the order in which the traits where incorporated. It is possible to define a different order (and a different link like AND or OR) for each method.

4.3.4 Composition of traits

The composite pattern requires that the composite unit behaves and looks from the outside just like the atomic unit (cf. (Olsen, 2008 S. 111-125)). Now, what is the atomic unit in this composition? The basic trait. What is the composite unit? The composed trait. But what is a trait? In a nutshell, it is a collection of methods.[31] What makes a trait with a class that it gets incorporated into? It adds methods to the class. The class itself also is a collection of methods.[32] So, basically, a trait adds a collection of methods to another collection of methods, called the incorporator. The added methods appear to be implemented in the incorporator itself. Now what happens if the incorporator is not a class but another trait? The incorporating trait gets more methods that appear to be implemented in the trait itself. When the resulting trait is used to be incorporated into

[30] This includes the incorporating class as well
[31] A precise definition is given in chapter 2.2
[32] A class also is a generator for instances and defines the state of objects.

another incorporator[33], those methods will simply be added to the incorporator. This behavior is exactly what the composite pattern demands.
This idea translates to ruby code very fluently as a class is just a special kind of module, which is just a collection of methods with a name. In ruby, a module is a collection of methods and a definition of state – the class only adds the aspect of generating instances.[34] This is very interesting as traits are defined with a module and classes – in the context of the implementation of traits can be seen as a special kind of module. So conceptually implementation wise, a trait adds methods to a module, not to a class – even if this module happens to be an instance of `Class`. The trait does not differentiate between classes and modules, so when a module happens to be the definition of a trait, that module can easily make use of the concept of traits to define its methods. In other words: A module that is used to define a trait can be built from traits just as any other module.
This idea is implemented with a module called `Traitable`. It can be included into anything that can be built from traits, thus classes and modules, which define a trait. In chapter 4.2 "Building classes with traits" there is an example of a class using `Traitable` and in appendix 11.5 "Composition of traits" a trait uses `Traitable`, therefore being composed of other traits.

4.3.5 Except and Only

Implementing Except and Only is not trivial using Rubies built-in mixin mechanic as mixins can only be included fully or not at all. It is not possible to partly include a mixin module. Thus it is not possible to simply map this feature to a function of mixins, a rather complicated approach was used to solve this problem.
Since Only is just an alternation of Except this chapter focuses in the implementation of Except. Only is derived from that by subtracting the defined methods from all methods and then using Except with the resulting set of methods.
The proposed implementation once again makes use of mixins and the ancestors to achieve the behavior of Except which is that all methods of a trait are incorporated into a class except those that are defined in the Except clause. Taking the defined Except methods, a new mixin is defined that does not implement any method by itself, it rather implements a method look up for the next mixin. This new mixin is called an ExceptFilter, as it filters for all defined methods and makes the method look up skip the next mixin in the ancestor chain. The Filter is included directly after the trait mixin so that its position in the ancestors is directly before the trait implementation of the method. Due to the implementation

[33] This incorporator might be a class or another trait.
[34] In this regard, a class is an extension to a module, as it adds the behavior of generating instances and inherits the rest of the functionality of a module – namely the definition of methods and the definition of state. However, a module can be seen as an extension to a trait as it inherits the functionality of the definition of methods from traits and adds the functionality of definition of state. If traits were built into ruby itself natively, this model might be used to express and implement the relation among trait, module and class. To illustrate this idea, a simple UML diagram is given in appendix 11.4. This idea is not core to this work or the proposed implementation.

within the filter the next mixin – the trait – is skipped and the method look up continues from the next but one.
There are several other approaches that promise to be easier to understand and to implement. Each of those has a critical downfall, ultimately leading to the rather complicated approach from above.

An easier approach would have been to **remove** the given method from the trait. However, since a trait is internally implemented with modules, the method would be removed from the module itself. So if a method would be excluded in one place it would automatically be excluded in the whole project. This problem could have been solved by copying all methods from the module into a new module so that when the excluded method is removed, it is only removed from this one module. Due to the open module principle described in chapter 3.2 "Open classes" it is possible to add methods to a module – and thus to the trait – after it was incorporated. This is dangerous in itself since the conflict detection is circumvented. But with the copied module, those later added methods would only be added to occurrences of the trait without Except and Only but not occurrences where Except or Only where used. This would lead to very unexpected and unmaintainable code. Thus the option of removing excluded methods from traits was not chosen.

In Ruby, there is the possibility to **unbind** method in one class and to rebind it to another class[35]. Therefore it would be possible to copy all methods from a module and to add them to the incorporating class. Methods that were added to the trait after it was incorporated into the class would not be added to that class, which is a much more expected behavior. However, there is one problem with this approach. The unbound method can only be rebound to the very class it came from or its sub classes. It is not possible to "trick" the `bind` method by modifying `kind_of?` or `is_a?` in the way that the target class appears to be a sub class of the method originator[36].
This approach of rebinding methods from the modules would have been conceptually cleaner, as modules would just have been used as a container of methods without using them as mixins. This would have let to a cleaner distinction between mixins and traits within the implementation. It is, not possible with the given meta programming techniques for reasons described in this paragraph. Further discussion about the possibilities of alternative implementations can be found in chapter 4.5 "Alternative implementations".

[35] Or to another eigenclass and therefore to a single object.
[36] This is due to the implementation of the bind method. It is not implemented in ruby itself but in C. The type check is hard coded into the interpreter and cannot be circumvented without modifying the interpreter itself. The appropriate code snippets are highlighted in appendix 11.9 "Code and documentation of the method bind"

4.3.6 Trait look up and extendable syntax

In the given examples traits where always referred by their name as a Symbol. The trait `MyGame::MyTraits::Position` would be referred to by `:position`. This is implemented just for convenience. Traits can also be referred by their full reference as the defining module or as the trait object itself.
The Symbol syntax triggers the trait look up which will look into certain modules that function as a name space and look for modules that are defined within that namespaces. These namespaces are defined in the constant `Traits::HOME` which is an array of modules. A game programmer only needs to append the namespace in which he defines his traits to enable the symbol syntax for his own traits.
Beyond that, not only Symbols and Strings could be used to refer a trait, the game programmer can add a `to_trait` method for any object or class and use it to reference traits in incorporation definitions.[37] This principle is attended to the `trait` class macro as well. The hash parameter just has a method `to_trait_incorporation` that is then sent to it. If the game programmer wishes to do so, he can easily change or extend the incorporation definition syntax.

4.4 Problems

The proposed implementation satisfies most of the requirements for traits, which are defined in this work. However, there are still some downfalls.

4.4.1 Unsatisfied methods

For one, there is no explicit enumeration of required methods, possibly leading to a programmer not knowing what the required methods of a trait are – e.g. when it is not documented. Also it is not possible to explicitly check for unsatisfied methods, as those will only raise an error the first time they are called.
An explicit listing of required methods would be possible via a constant with a certain name or with a module macro[38].

4.4.2 Clean methods

The requirement of "clean methods", thus methods that cannot access state directly, is not checked either. Traits are defined with a module, but within modules the state of an object can be accessed directly. There is no direct way to check whether a method uses instance variables or not, as the interpreter allows that. One possible solution might be to parse the source code of the module that defines the trait. This is no trivial task especially due to the dynamic nature of ruby. Therefore, and due to time-wise constrictions, it is left to documentation and the trait programmer to not access state directly.

[37] In fact, the trait lookup is simply triggered by `Symbol#to_trait` and `String#to_trait`.
[38] Module macros are described in chapter 3.6 "Class and module Macros"

4.4.3 Open module

As described in 3.2 “Open classes” classes and modules can be reopened at any time to extend or overwrite their functionalities. When a class includes a mixin module, it is extended with the methods within that module. When that module is later opened and a method is added, the class will also know that method. For mixins this is a neat feature, for traits, there occur problems. Name conflicts among methods are checked upon incorporation. When there are later methods added to that module, it might happen that there are undetected method conflicts. This behavior is left to be tested and experimented with. A possible solution might be to use the hook method `Module#method_added` to forbid adding methods after the module has been used for trait definition. This however, is a strong restriction that would need to be considered wisely, as traits could also benefit from open modules as long as the name conflicts are detected and solved in a symmetric way.

4.4.4 Performance

Performance was not a top priority during the design of the proposed implementation. A small benchmark about the creation of 50000 classes showed that a class definition with traits is 50 times slower than the “equivalent” class definition with mixins.[39] This is not too concerning as class definition is only done once at the beginning of the runtime of a program. More interesting are the time costs for calling methods that are composed of traits and methods that are composed of traits. As it turns out, the method look up and execution for resolved methods from traits is about 17% slower than its equivalent mixin method.[40]

4.4.5 Semantic name conflicts

A rather conceptual problem with traits can occur, when traits are composed. Assuming that there is a trait called `Emotion` that models the emotion of a non-player-character in a game and another trait `Movable` that is responsible for moving that non-player-character on the screen, both of them can implement the method `moved?` but both of these implementations have a totally different semantic. This conflict can be solved by the `NonPlayerCharacter` class that incorporates both traits. The problem arises when there is later on another trait `AdvancedEmotion`, that composes `Emotion` and provides more advanced methods regarding emotion. The programmer of `AdvancedEmotion` might assume that the message `moved?` will execute the `moved?` method in `Emotion`, as he cannot know all other traits that might be incorporated into the class that `AdvancedEmotion` is incorporated into. He might write a method like this:

[39] The program code and the result of that benchmark can be found in appendix 11.6 "Class definition benchmark”

[40] See appendix 11.8 “Method call benchmark”

```
1 def show_emotion
2   if moved?
3     shed_tear
4   end
5 end
```

Fig 22: An example for semantic name conflicts

If the programmer of the class decides to solve the `moved?` conflict in that regard, that he just picks the `Movable#moved?` and ignores the `Emotion#moved?`, this `show_emotion` method would lead to the behavior that the non-player-character sheds tears whenever it is moved on the screen.
A solution to this problem would be to just call `moved_in_emotion?` in line 2. This however, would only work if there is a name conflict with the method `moved?`. To prevent such errors the programmer of `AdvancedEmotion` could check with `respond_to?` whether a `moved_in_emotion?` is defined and only then call it and otherwise just call `moved?`. This would be a rather inconvenient solution to the problem.
A more simple solution that leads to more readable code would be to alias all methods whether they are conflicted or not. Then the programmer of `AdvancedEmotion` could always write `moved_in_emotion?` and be sure the right method is called. The pros and cons of that approach are discussed in chapter 4.3.2 “Conflict solving”.

4.4.6 Super

The flattening characteristic requires that the methods, which are provided by a trait, can be seen as if they were copied and pasted into the class. This especially must apply to the behavior of the keyword `super`. In the case of ruby `super` would execute the implementation of the mixin that was at last included or if no mixin was included it would execute the implementation of the super class.
This behavior was not achieved in the given implementation as traits are implemented via mixins that are part of the normal method look up of ruby. In the time that was invested into this work,no fitting solution was found. A possible solution would be to modify super in a way that it skips those mixins that are used as traits. This would not be possible without modifying the interpreter itself, which is discussed in chapter 4.5 “Alternative implementations”.

4.4.7 Flattening precedencies violated

As described in chapter 4.3.1 "Conflict detection” the instance methods of the incorporator are included in the conflict detection. So if a class implements the `update` method and incorporates a trait that implements the `update` method, this method will be considered a conflicted method in the given implementation. This behavior violates the flattening characteristic described in chapter 2.2.6 ”Flattening traits into a class”. In that chapter it is

defined that a method implemented in the class takes precedence over the method provided by a trait.
This variance was implemented on purpose as it was not regarded as a core characteristic of traits. It was rather regarded as a discrepancy to the explicit conflict solving characteristic as it implicitly solves the conflict between trait and incorporator. Of course, it is not contradiction in the definition because conflicts between a trait and its incorporator are not defined as a conflict and therefore does not need to be solved explicitly.
However, in practical terms, the author was not of the opinion that this implicit resolve would lead to better understanding of the trait concept. Also a programmer might not know all the methods provided by a trait and unintentionally overwrite a method that is used by the trait itself. This would lead to a semantic name conflict[41] after all. Beyond that, the exception in conflict detection was considered a breach in uniformity among traits and incorporators.

4.5 Alternative implementations with more than plain ruby syntax

With the given approach of implementation, it can be seen that there are several problems left to improve on. Some of which may be easier to fix while others might pose more severe challenges.
It is clear that there are limits to what can be done with meta programming techniques in ruby due to its first-class-ness. While many Objects in ruby are first-class citizens there are still some things that are not reachable from within ruby, namely the interpreter itself, the call stack, assignments and instance creation to name a few.[42]
Without these limitations it would have been possible to introduce a new language artifact trait with its own keyword just like class or module only using pure ruby syntax. A possible concept of such an artifact is shown in appendix 11.4 “The hypothetical relation among Trait, Module and Class”.
In the given circumstances, the introduction of a new keyword is only possible when modifying the interpreter that is written in C itself. It would also be thinkable to change the way method lookup works, modifying the interpreter. This is even possible – to a certain extend - using only meta programming by overwriting the method `send`.

[41] Semantic name conflicts are described in chapter 4.4.5 “Semantic name conflicts”.
[42] To the knowledge of the author, the project rubinius tries to solve these problems by implementing ruby in ruby, thus making it more first-class. However, during the course of this work the proposed implementation was designed for the standard ruby implementation, also called MRI for matz ruby interpreter – named after rubies inventor Yukihiro Matsumoto (cf. (Ruby community)).

It was foreseen to change the interpreter as one of the core aspects of this work is how traits can be implemented using only meta programming techniques of ruby.[43] This self-limitation was raised due to two main reasons:

- To measure capabilities of rubies meta programming techniques and
- To make the proposed trait implementation compatible to as many different ruby implementations as possible.

[43] Also, there are other reasons the plain problem of complexity when dealing with interpreter plugins. In addition to that it would have been too time consuming to earnestly consider such implementation.

5 Applying traits to a game engine

In chapter 2 “Traits”, traits have been defined while in chapter 4 “Extending ruby with traits” an implementation was proposed. In this chapter 5, there will be shown how traits could be applied to a game engine. To do so, Chingu[44] was chosen, as it is a 2D game engine that is fully written in ruby, based on Gosu[45], which is written in Ruby and C++.

5.1 Chingu

This chapter gives a short introduction to Chingu. It highlights important design artifacts but leaves some things behind. A more complete introduction to Chingu can be found in the Chingu documentation[46].

5.1.1 Game loop

As many other game engines as well, Chingu functions in terms of game loop. Within a game loop all the logical behavior of the game as well all the rendering is computed. It is triggered at regular intervals dependent from the target frames per second. Of course the game loop only starts after the engine is initialized and resources are loaded. Resources can be loaded as a reaction to player input during the game loop, as well.[47]
To distinguish even more between game logic and rendering logic, the game loop is divided into `update` and `draw`, which are also methods in the main window.

[44] The official website for Chingu is http://ippa.se/chingu
[45] The official website for Gosu is http://www.libgosu.org/
[46] Chingu documentation: http://rdoc.info/github/ippa/chingu
[47] In fact, in some games it makes more sense to load resources during the game play. If there is a huge world it is not wise to load all assets into the memory when most of them won’t be needed. It is advisable to load only assets that are near the player.
This mechanic is often implemented with levels, where there is a loading phase at the beginning of the level.

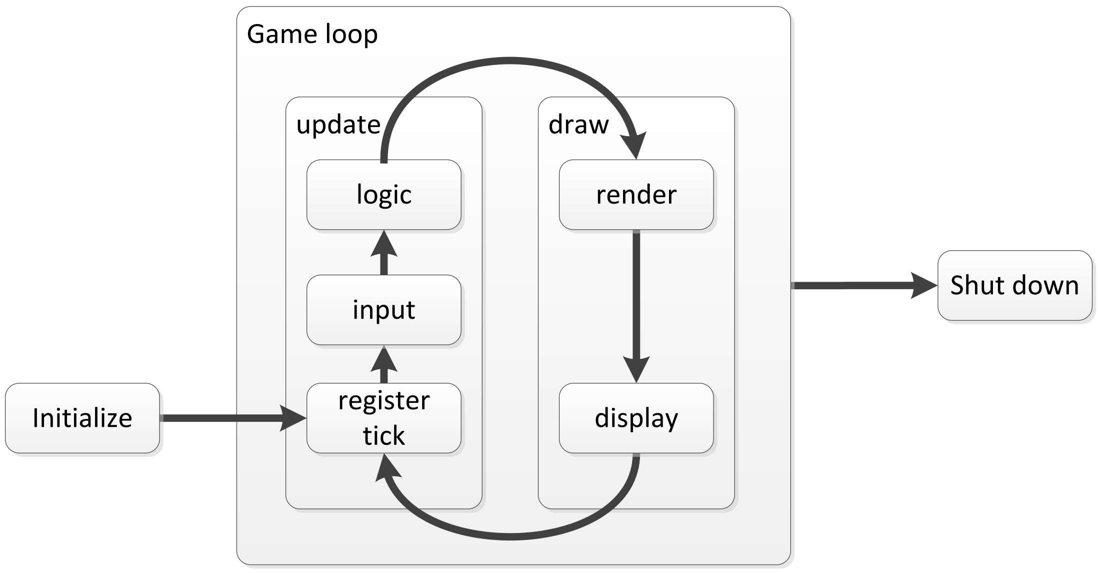


Fig 23: The processes that happen during the execution of a game can be divided into initialize, game loop and shut down. The game loop again is divided into update and draw that are divided again. Within the step "logic" there might be features like acceleration, relocation, collision detection, planning, pathing, etc.

5.1.2 Game objects

A powerful concept of Chingu, which is also used in many other game engines, is the game object. It is a class that is supposed to define common behavior of "anything that is in the game". In Chingu, each game object can be drawn on the window, can be updated and can receive input from the player. It is worth of discussion whether all game objects need to have these aspects[48] as only controllable objects like the player character or menus need to receive player input. Also only manipulated objects need to be updated, e.g. a rock that is just an obstacle and cannot be modified in any way.

5.1.3 Game states

Games usually can be seen as a state machine. In the beginning there is a main menu, then a level starts, maybe a world is entered, a fight begins, a trade is being done, etc. During all these phases of a game it can behave totally different. Especially input can mean something totally different. While pressing A on the gamepad during a trade means "buy the potion", during a fight, it may mean "jump". But also the drawing aspect of the game can change due to a game state. Maybe, upon death, the world is greyed out, or simply in the main menu, the player character is not shown.

[48] Aspect is used in a common sense meaning, not in the software meaning.

To make managing those states of a game easier, Chingu offers a dedicated game state model with a stack-based game state management. There are also transitional game states and hook methods for each game state when pushed and popped.

5.1.4 Traits

"The aim [of traits] is so [sic!] encapsulate common behavior into modules for easy inclusion in your game classes. Making a trait is easy, just an ordinary module with the methods setup_trait(), update_trait() and/or draw_trait()." – Chingu README (Ippa, 2012)

So Chingu also has a concept that is called trait. However, a closer look at the implementation of Chingu traits, which can be seen in appendix 11.7, reveals that Chingu traits are rather mixins than traits, like they are defined in this work. This is even mentioned in the Chingu README: "Chingus trait-implementation is just ordinary ruby modules with 3 special methods: setup_trait, update_trait, draw_trait" (Ippa, 2012). The author of Chingu also is aware of the problem with linear composition of mixins as he writes "Each of those 3 methods must call 'super' to continue the trait-chain." (Ippa, 2012)
Chingu traits do not only implement those three above methods, they can also implement any other method like any mixin can. In addition to that they can have an inner module `ClassMethods` to define methods that will be added to the including class. Chingu traits can also be included in the context of certain options that are custom to the trait. A trait Acceleration could have an option DEBUG to draw the acceleration as vector when the DEBUG option is true. These options are bound to the including class, so either for all instances of a certain game class there would be drawn that acceleration vector or for none.
Chingu not only offers the trait model but also many implementations of traits such as `animation`, `collision_detection`, `sprite`, `velocity` and `timer`. The trait model is even used to structure Chingus inner code, the `GameObject` class, for example, consists only of the Chingu trait `sprite` and the ruby mixin `InputClient`.

5.2 Discussion on Chingus trait model

The downfalls of mixin inheritance are already described in chapter 2.1.5 "Mixin inheritance" and are therefore not discussed here. But apart from inherent characteristics of mixin inheritance there are some specialties to Chingu traits.

5.2.1 Class methods

One of them is the `ClassMethods` module that extends the class that includes the Chingu trait. It is heavily used in the trait `collision_detection` to enable the programmer not to only check for collision among certain objects but among whole groups of objects. An example for this is in the documentation of

`Chingu::Traits:CollisionDetection::ClassMethods#each_collision`

```
Enemy.each_collision(Bullet).each do
  |enemy, bullet| enemy.die!
end
```

The addition of a definition of class-level methods within a trait does not interfere with the trait definition in this work and is therefore conceptual compatible. It adds more expressiveness to traits themselves as well as to the classes that use those traits. The proposed implementation of traits does not have such a feature explicitly. However, as traits can be applied to any class, they can be applied to eigenclasses as well. Therefore they can be used to define class-level methods as well. This approach is not as convenient and intuitive as the class-level method definition of Chingu traits, though.

5.2.2 Options

Chingu traits can also have options for the inclusion, therefore parameterizing their behavior for each class. This also adds to the flexibility and expressiveness of traits, for the DEBUG example in chapter 5.1.4 “Traits” it is however required to store those options in a class variable. In fact, most options that are used within the Chingu traits, store the options in a class variable to access them later. This handling with options is not compatible with the definition of traits in this work. Yet still the proposed implementation in this work enables options, although only in the very limited range of `:except` and `:only`, both of which do not require state access.

5.2.3 Applications

In Chingu the implementation of the trait method is the core of the implementation of traits themselves. There is no dedicated class or mixin module to implement this feature. This is not necessarily a bad design decision, if there was only place where traits are defined. However, in Chingu there are two places where traits can be applied: `GameObject` and `GameState`. Both classes have 100% redundant implementations of the same method. If something other than game objects or game states needs to make use of Chingu traits there is no convenient way to make that happen. In fact, the trait `timer` is a good example for this downfall. The trait timer offers methods like `after(millis)` or `during(millis)` which receive a block and execute them after or during a certain time span. This behavior is not necessarily bound to a game object or game state. E.g. `Chingu::Window` is also a class that could take great use of timer logic. In contrast to that, the proposed implementation of traits is applicable to any class, module or trait – thus offering more flexibility and extensibility.

Table 6: Appraisal and comparison of Chingus trait model

Appraisal and comparison of Chingus trait model		
Aspect	**Appraisal of Chingus model**	**Proposed implementation**
Consistency with the definition in this work	Traits, as implemented in Chingu, do not fulfill the requirements of traits as defined in this work. They are just mixins with some extra features.	Mostly consistent with the definition in this work.
ClassMethods	Useful feature that increases expressiveness.	No dedicated feature. Traits can be applied to eigenclasses to compensate for that.
Options	Custom options for each trait.	Only `:except` and `:only`
Options storage	Stored within class variables and then accessed during the lifetime of the instances.	Transient; Only needed during the process of incorporation; No Storage
Applicable to	`GameObject` and `GameState`	Any class, module or trait

5.3 Application of new trait implementation

As seen in the above Table 6: Appraisal and comparison of Chingus trait model there are some differences between the trait model offered by Chingu and the trait model offered by this work. Now, to make Chingu benefit from the trait implementation presented in chapter 4.3 “Implementation of traits”, some changes need to be made to the implementation of Chingu. The actual implementation of the offered trait implementation to Chingu is not part of this work due to time issues. It is comprehended on a theoretical basis.
First of all, the redundant implementations of `trait` and `traits` in `GameObject` and `GameState` need to be removed. Those two classes need to include the mixin module `Traitable`, instead.
The methods `update_trait`, `draw_trait` and `setup_trait` are then not needed any more. They can be completely removed as there is now a collision detection for naming conflicts. This makes code easier to understand, as there is no logical difference between `update` and `update_trait` and now there is no difference in the code as well.
In a last step, the traits themselves need to be rewritten in a form that they are compatible with the trait definition in chapter 4.1 “Trait definition”. Again, the `update_trait` etc. methods need to be renamed. A more complex problem poses the `ClassMethod` modules, which add methods to the class level. There is no direct way to map this functionality to the new trait implementation.

5.4 Decomposition into traits and composition from traits

When dealing with an existing class hierarchy it can be hard to find the right approach to apply traits. All existing classes are complete but not as coherent as traits would be. The software designer has to decide what code is worth a refactoring into traits. Code has to be cut out of classes and then reincorporated into the class via traits. This process is called decomposition into traits and composition from traits.
As it is hard to find appropriate tools to find possible approaches to integrate traits, namely duplicated code (cf. (Murphy-Hill, et al., 2005)), it may be a valid strategy to have a more abstract look at the classes at hand and their responsibilities. This however is a whole new field of research that is not core to this work.

5.5 Comparison of three implementations of Monster

An example of a game is given, where the player has to kill several monsters to survive. He can lay bombs to destroy rocks and find power ups. What behavior defines an instance of the class `Monster`? How is it written in Code? In this chapter there are three versions of the `Monster` class:

- Without any Traits, just single inheritance
- With Chingus trait model
- With the proposed trait implementation

```
class Monster < Chingu::GameObject # Monster with single inheritance

  attr_accessor :planning, :path_finding, :positioning,
                :bounding_box, :collision_detection # Delegate objects

  def setup
    # Initialize delegate objects...
    self.planning            ||= Planning.new
    self.path_finding        ||= Pathfinding.new
    self.positioning         ||= Positioning.new(position:     [0, 0],
                                                 speed:        [0, 0],
                                                 acceleration: [0, 0])
    self.bounding_box        ||= BoundingBox.new(width:  graphics.width,
                                                 height: graphics.height)
    self.collision_detection ||= CollisionDetection.new(shape: bounding_box)
  end

  def update
    planning.update(path_finding)
    positioning.update
    collision_detection.update
  end
end
```

Fig 24: When monster does only benefit from single inheritance, there are only two options to add features to the class without implementing them within the class itself: Inheritance and delegation. In this case the code for graphics and for input are inherited from GameObject, the code for planning, path finding, positioning and collision detection is added via delegate objects.

It is noted that the options for default state values are added on the initialization of the delegate objects. The conflict must be resolved within the instance method `update`. Also there are two features inherited from `GameObject`: Graphics and input. Since Monster is not controlled by the player whatsoever, there is no need for any code in Monster that handles input deferral. It becomes obvious that GameObject is not fine grained enough to offer an efficiently reusable set of features.

```
 1 class Monster < Chingu::GameObject # Monster with chingu traits
 2
 3   trait :vectors, { position:     [0, 0],
 4                     speed:        [0, 0],
 5                     acceleration: [0, 0] } # Position
 6   trait :timer                             # Timer methods
 7   trait :bounding_box                      # Needed for collision detection
 8   trait :collision_detection               # Collision detection
 9
10   attr_accessor :planning, :path_finding   # Delegate objects
11
12   def setup
13     # Initialize delegate objects...
14     self.planning     ||= Planning.new
15     self.path_finding ||= Pathfinding.new
16   end
17
18   def update
19     planning.update(path_finding)
20   end
21 end
```

Fig 25: In this case, Monster is implemented with chingus traits. The options for default state values are defined upon incorporation (see line 3). Still there is need for some delegate objects, this however is only the case due to the lack of a build-in trait for planning.

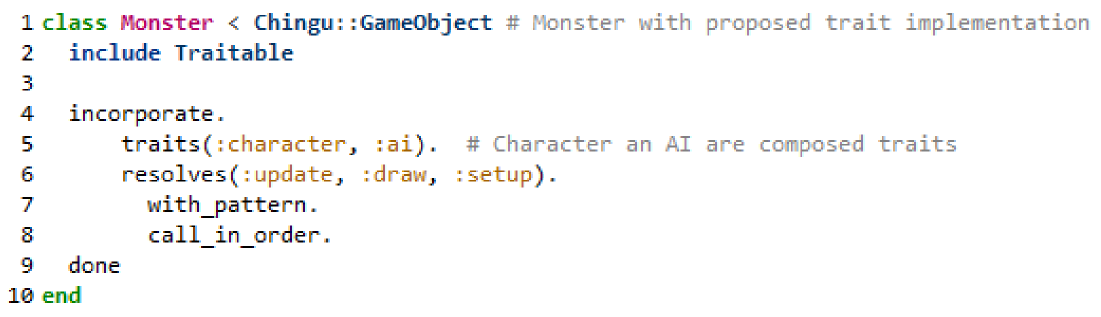
While this code seems more elegant, there are still problems that are rather subtle. There is no need to explicitly resolve `update` as they are combined using the super chain in chapter 2.1.5 "Mixin inheritance". Each mixin calls super, thus triggering the next implementation of `update`. Originally the last mixin may not call super as there is no next implementation, but since the programmer of the mixin does not know whether it is the last mixin or not, there has to be a super. This leads to a solution where the super class has an empty implementation of `update` that is called be the last mixins `super`. This can be envisioned with a look at this excerpt from `BasicGameObject`.

```
    #
    # Empty placeholders to be overridden
    #
    def self.initialize_trait(options); end
    def setup_trait(options); end
    def setup; end
    def update_trait; end
    def draw_trait; end
    def update; end
    def draw; end
```

Fig 26: BasicGameObject implements empty placeholder of different methods so the mixins can combine their implementations via super.

While this approach seems valid to combine the behavior of the above methods there are still questions unanswered.

- What if a programmer of a trait wants to user other methods than `setup`, `draw` or `update`?
    - For each new method in any trait there would need to be added a new empty placeholder in `BasicGameObject`.
- What if the new methods have an actual return value?
    - For the methods so far the return values can be ignored. It is just important to call them in a certain order. What if there are methods that return Boolean values? Will they be linked via AND or via OR? The programmer of each trait has to decide and hard code it into the trait. The programmer of the class has no chance but to accept the decision made in the trait.
- What if not only the kind of link between return values is important but also the ordering?
    - For Boolean values there is no difference in which order they are linked via AND or OR[49]. But for example a String concatenation is not commutative. The programmer of the class might wish to determine the order in which the different String return values are concatenated. This is another phrasing of the problem with asymmetric composition or linearization described in chapter 2.2.3 "Symmetric composition".



```
class Monster < Chingu::GameObject # Monster with proposed trait implementation
  include Traitable

  incorporate.
      traits(:character, :ai).  # Character an AI are composed traits
      resolves(:update, :draw, :setup).
        with_pattern.
        call_in_order.
  done
end
```

Fig 27: With the proposed trait implementation the Monster class becomes considerably shorter. Also the super class GameObject does not offer any features mentioned in this example, it can be considered a markup.

The shortage of the code can be ascribed to the composition of traits. Character and AI are traits that are composed of other traits, which in the end results in the behavior of the other two Monster classes shown before. In this case the ordering and linking operator among the different versions of update is not important. Thus the pattern `call_in_order` is used to keep the code as simple and readable as possible. The problems described for the second Monster class could be circumvented by using the explicit resolve variant using a lambda and aliased methods.

[49] Other than performance.

6 Conclusion and future work

6.1 Conclusion

This work has given a definition of traits – a language artifact that is used to build classes in a fashion that each trait can be reused easily for other classes, thus increasing modularity. It is shown that traits are more fine grained and therefore can achieve more reusable code where inheritance or mixins still have conceptual problems. Those are mainly the linearization, which means the order in which classes inherit from one another and the order in which mixins are included into classes are important. The behavior changes with the changes in that order. Also the programmer of such classes and mixins need to know where they are within the hierarchy due to the keyword super. The knowledge of the position within the class hierarchy leaks into the code of the respective code entities. This decreases the reusability of such classes and mixins drastically. Traits solve this problem by being commutative, which means symmetrically composable. The explicit conflict solving for overloaded methods immensely helps achieving this behavior.

A brief introduction to rubies meta programming techniques was given so the reader can comprehend how traits are implemented in the proposal of the author. It is described how traits can be implemented specifically in ruby only using ruby syntax. This approach has the advantage that it is compatible with every correct ruby implementation but also has its boundaries due to the limited first-class-ness of ruby. The author shows how traits can be implemented based on mixins, by using aliased versions of methods and explicit conflict detection.

Finally it is shown how the proposed trait implementation could be applied to the game engine Chingu. Chingu already has a concept that is called traits, but that is rather a mixin with some engine specific features. First of all, those Chingu traits would need to be replaced by traits as defined in this work. After that the already defined Chingu traits could be used to compose higher level traits to minimize the effort the user of the engine has to make to define his game object classes. An example is given how such a class can benefit from the authors trait implementation compared to the mixin-like trait implementation of Chingu itself.

6.2 Future work

During the work with Chingu and its trait model it became clear that many traits are bound to some kind of state, often a set of instance variables. While Chingu traits are free to define these variables themselves and even initialize them, the proposed implementation only allows pure methods as mentioned in the definition in chapter 2.2.1 “Pure methods”.

This restriction was defined due to the sake of simplicity in the theory of traits. In practice, however, it would be handy to drop this restriction to make traits more comfortable to use. Instead of identifying all the getters and setters that a trait needs to function it just defines them itself. This on the other hand would define the separation of the state and the behavior of an object down. To keep this separation clear it might be worthwhile to have a new concept to define the state of an instance in the fashion of traits. Thus not only the behavior can be used with fine grained reusable chunks of code, but also the state. This concept might be called *part state*.

Without any further exploration of this idea, the author came to the impression that the case of conflicting methods in different traits is a rather unlikely one. In most cases, the class programmer might not need to define any resolves at all. In these cases, it might be convenient to have a much more compact syntax to describe the incorporation of traits into a class. The following syntax might be appropriate:

```
def_class(:Monster,[:health,:position,:speed],[:moving,:shootable,:position,:acceleration])
```

Fig 28: A possible syntax to build a class from traits when there are no conflicts. In this case `health`, `position` and `speed` are the instance variables while `moving`, `shootable`, `position` and `acceleration` are the names of traits.

In (Ducasse, et al., 2006) there is a class browser described that can view classes as if they were flatly implemented only with methods. All incorporated traits are virtually copied and pasted into the class. It is suggested that such a tool helps developing classes with traits and makes understanding them easier. In (Murphy-Hill, et al., 2005) it is described that finding approaches for applying traits to an existing class hierarchy is hard even with dedicated tool support. This leads to the conclusion that developing a tool that helps dealing with traits might be a worthwhile field of further research.

Another obvious field of further research is to approach the problems stated in chapter 4.4 "Problems”.

7 Annotations

The source code of the proposed implementation is available at https://github.com/AKnopf/trait
It was developed and tested on/with:

- Windows 7
- MRI 1.9.3p194 [i386-mingw32]
- RubyMine IDE 4.5.4

8 Source materials

Alexandrescu, Andrei. 2000. *Traits: The else-if-then of Types.* 2000. http://erdani.com/publications/traits.html.
Costanza, Pascal. 2003. Dynamically Scoped Functions as the Essence of AOP. [Paper]. Bonn : University of Bonn, 6 17, 2003. http://dl.acm.org/citation.cfm?id=944587.
Ducasse, Stephane, et al. 2006. *Traits: A Mechanism for Fine-Grained Reuse.* NY, USA : ACM, 2006.
Durr, Pascal, Bergmans, Lodewijk and Aksit, Mehmet. 2007. *Reasoning about Behavioral Conflicts between Aspects.* s.l. : University of Twente, The Netherlands, 2007.
Feigenspan, Janet. 2008. The Diversity of the Understanding of Aspect-Oriented Programming. Magdeburg : University of Magdeburg,, 2008.
Filman, Robert E. 2001. *What Is Aspect-Oriented Programming, Revisited.* Budapest : s.n., 2001.
Hirschberger, Johannes. 2007. *Geschichte der Philosophie.* Freiburg im Breisgau : Komet, 2007. Vol. Band 1.
Ippa. 2012. Rdoc.info. [Online] Ippa, 5 23, 2012. [Cited: 09 07, 2012.] http://rdoc.info/github/ippa/chingu#Traits.
Jarvis, Lee. 2012. Rubydoc.info. [Online] 11 2, 2012. [Cited: 2 11, 2012.] http://rubydoc.info/gems/traited/1.0.0/frames.
Malayeri, Donna. 2008. *CZ: Multiple Inheritance Without Diamonds.* Nashville, Tennessee, USA : OOPSLA'08, 10 19, 2008.
Murphy-Hill, Emerson R., Quitslund, Philip J. and Black, Andrew P. 2005. *Removing Duplication from java.io: a Case Study using Traits.* Ney York : ACM, 2005.
Odersky, Martin. Scala-Lang.org. [Online] [Cited: 10 16, 2012.] http://www.scala-lang.org/docu/files/ScalaReference.pdf.
Olsen, Russ. 2008. *Design Patterns in Ruby.* Upper Saddle River, NJ : Addison-Wesley, 2008. p. 111pp.
Perotta, Paolo. 2010. 1.2 Open Classes. *Metaprogramming Ruby.* Raleigh, North Carolina; Dallas, Texas : Pragmatic Bookshelf, 2010, p. 261.
—. 2010. 4.6 Aliases. *Metaprogramming Ruby.* Raleigh, North Carolina; Dallas, Texas : Pragmatic Bookshelf, 2010, p. 261.
Perrotta, Paolo. 2010. 1.5 What Happens When You Call a Method? *Metaprogramming Ruby.* Raleigh, North Carolina; Dallas, Texas : Pragmatic Bookshelf, 2010, p. 261.
—. 2010. *Metaprogramming Ruby.* Raleigh, North Carolina; Dellas, Texas : Pragmatic Bookshelf, 2010. p. 261.
Rojas, Raúl. 1998. *A Tutorial Introduction to the Lambda Calculus.* Berlin : FU Berlin, 1998.

Ruby community. ruby-lang.org. [Online] [Cited: 11 27, 2012.] http://www.ruby-lang.org/de/downloads/.

ruby-doc.org. ruby-doc.org. [Online] [Cited: 11 12, 2012.] http://www.ruby-doc.org/core-1.9.3/Proc.html#method-i-lambda-3F.

Schärli, Nathanael and Black, Andrew P. 2003. *A Browser for Incremental Programming.* Beaverton, Oregan, USA : OGI School of Science & Engineering; Oregon Health & Science University, 2003.

Schärli, Nathanael, et al. 2003. *Traits: Composable Units of Behaviour.* [prod.] University of Bern, Switzerland Software Composition Group. Bern, Swizerland : Springer Verlag, 2003.

Schärli, Nathanael, et al. 2002. *Traits: The Formal Model.* Bern, Switzerland : University of Bern, Switzerland, 2002.

Wikipedia. 2012. wikipedia.org. [Online] wikimedia, 6 28, 2012. [Cited: 10 16, 2012.] http://de.wikipedia.org/wiki/Trait_(Programmierung).

—. 2012. wikipedia.org. [Online] wikimedia, 9 3, 2012. [Cited: 10 17, 2012.] http://en.wikipedia.org/wiki/First-class_citizen.

Wunderlich, Lars. 2005. 1.7.6 Symmetrische und asymmtrische AOP-Lösungen. *AOP - Aspektorientierte Programmierung in der Praxis.* Frakfurt : entwickler.press, 2005, p. 288.

9 Figures

10 Tables

11 Appendix

11.1 Code example for composition of mixin modules

```ruby
module Position # Mixin with no composition
  def move_to(position)
    @position = position
  end

  def move_relative(vector)
    @position += vector
  end
end

module Speed # Mixin with composition for method update
  def speed=(vector)
    @speed = vector
  end

  def update
    move_relative(@speed)
    super
  end
end

module Acceleration # Mixin with composition for method update
  def acceleration=(vector)
    @acceleration = vector
  end

  def update
    self.speed=(@acceleration)
    super
  end
end

class FallingStone # Class that includes the three Mixins
  include Acceleration
  include Speed
  include Position
  # This order of inclusions leads to the ancestors:
  # [Stone, Position, Speed, Acceleration, Object, Kernel, BasicObject]
  # Thus Speed#update would overwrite Acceleration#update, however it is
  # accessed through super-call in Speed#update

  def initialize
    @position     = Vector[0, 0]
    @speed        = Vector[0, 0]
    @acceleration = Vector[0, 10]
  end
end
```

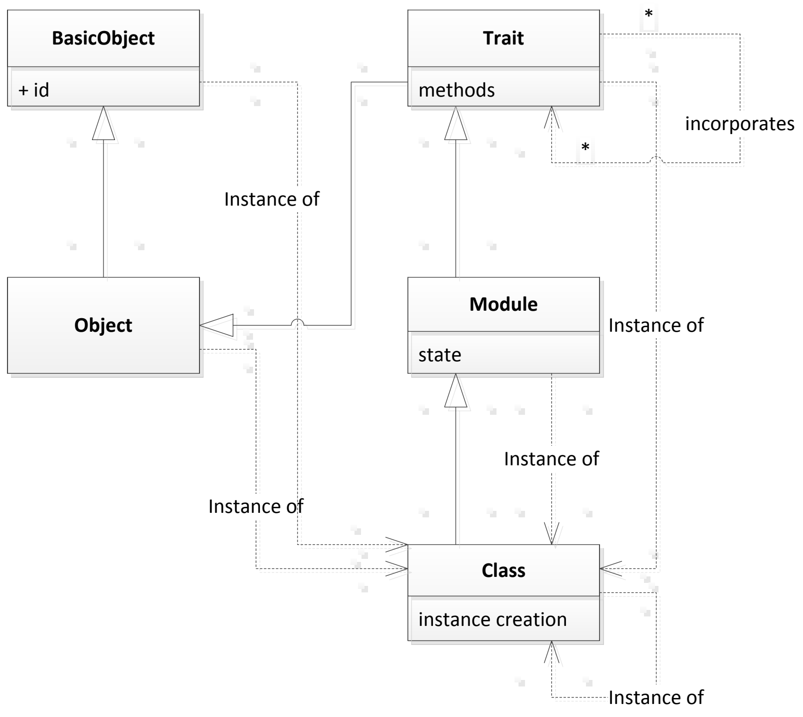
11.2 All methods of a String object

```
"A string".methods.inspect #  => [:<=>, :==, :===, :eql?, :hash, :casecmp, :+, :*,
                           # :%, :[], :[]=, :insert, :length, :size, :bytesize,
                           #:empty?, :=~, :match, :succ, :succ!, :next, :next!,
                           #:upto, :index, :rindex, :replace, :clear, :chr, :getbyte,
                           # :setbyte, :byteslice, :to_i, :to_f, :to_s, :to_str,
                           #:inspect, :dump, :upcase, :downcase, :capitalize,
                           #:swapcase, :upcase!, :downcase!, :capitalize!, :swapcase!,
                           # :hex, :oct, :split, :lines, :bytes, :chars, :codepoints,
                           # :reverse, :reverse!, :concat, :<<, :prepend, :crypt,
                           # :intern, :to_sym, :ord, :include?, :start_with?,
                           #:end_with?, :scan, :ljust, :rjust, :center, :sub, :gsub,
                           # :chop, :chomp, :strip, :lstrip, :rstrip, :sub!, :gsub!,
                           # :chop!, :chomp!, :strip!, :lstrip!, :rstrip!, :tr, :tr_s,
                           # :delete, :squeeze, :count, :tr!, :tr_s!, :delete!,
                           #:squeeze!, :each_line, :each_byte, :each_char, :each_codepoint,
                           # :sum, :slice, :slice!, :partition, :rpartition, :encoding,
                           # :force_encoding, :valid_encoding?, :ascii_only?, :unpack,
                           # :encode, :encode!, :to_r, :to_c, :>, :>=, :<, :<=, :between?,
                           # :nil?, :!~, :class, :singleton_class, :clone, :dup,
                           # :initialize_dup, :initialize_clone, :taint, :tainted?,
                           # :untaint, :untrust, :untrusted?, :trust, :freeze, :frozen?,
                           # :methods, :singleton_methods, :protected_methods,
                           #:private_methods, :public_methods, :instance_variables,
                           # :instance_variable_get, :instance_variable_set,
                           #:instance_variable_defined?, :instance_of?, :kind_of?, :is_a?,
                           # :tap, :send, :public_send, :respond_to?, :respond_to_missing?,
                           # :extend, :display, :method, :public_method,
                           # :define_singleton_method,:object_id, :to_enum, :enum_for,
                           # :equal?, :!, :!=, :instance_eval, :instance_exec,
                           # :__send__, :__id__]
```

11.3 Building a class with the class macro “trait”

```ruby
class Monster
  include Traitable
    trait(traits:     { movable: { except: :to_s },
                        scalable: {},
                        hittable: {},
                        shooting: {}},
          incorporator: self,
          resolves:     {moved?: lambda{ raise "Overwrite method 'Monster#moved?' to resolve"},
                        to_s: lambda{ to_s_in_scalable + to_s_in_hittable + to_s_in_shooting}})
end
```

11.4 The hypothetical relation among Trait, Module and Class



11.5 Composition of traits

```ruby
class Player
  attr_accessor :x, :y, :name, :speed_x, :speed_y, :acc_x, :acc_y

  include Traitable
  incorporate.trait(:acceleration).done
end

module Traits
  module Acceleration

    include Traitable
    incorporate.trait(:movable).done

    def acc
      Vector[acc_x,acc_y]
    end

    def acc=(x,y)
      acc_x = x
      acc_y = y
    end

    def update_speed
      self.speed = acc_x, acc_y
    end
  end


  module Movable
    include Traitable
    incorporate.trait(:position).done

    def speed
      Vector[speed_x,speed_y]
    end

    def speed=(x,y)
      speed_x = x
      speed_y = y
    end

    def update_position
      move_rel(speed_x, speed_y)
    end

  end


  module Position
    def move_to(x,y)
      self.x = x
      self.y = y
    end

    def move_rel(x,y)
      self.x += x
      self.y += y
    end
  end
end
```

11.6 Class definition benchmark

Benchmark:

```ruby
module Traits
  module Hittable
    def update
      "update in hittable"
    end
  end

  module Position
    def update
      "update in position"
    end
  end
end
n = 50000
Benchmark.bm(20) do |b|
  b.report('Define class with mixin: ') do
    n.times do
      Class.new do
        include Traits::Hittable
        include Traits::Position
      end
    end
  end
  b.report('Define class with trait: ') do
    n.times do
      Class.new do
        include Traits::Traitable
        incorporate.traits(:hittable, :position)
        .and
        .resolve(:update)
        .with_pattern
        .call_in_order
        .done
      end
    end
  end
end
```

Output:

```
                           user     system      total        real
Define class with mixin:   0.234000   0.000000   0.234000 (  0.252930)
Define class with trait:  11.954000   0.015000  11.969000 ( 12.037110)
```

11.7 Chingus trait implementation

```
def self.trait(trait, options = {})

  if trait.is_a?(::Symbol) || trait.is_a?(::String)
    begin
      # Convert user-given symbol (eg. :timer) to a Module (eg. Chingu::Traits::Timer)
      mod = Chingu::Traits.const_get(Chingu::Inflector.camelize(trait))

      # Include the module, which will add the containing methods as instance methods
      include mod

      # Does sub-module "ClessMethods" exists?
      # (eg: Chingu::Traits::Timer::ClassMethods)
      if mod.const_defined?("ClassMethods")
        # Add methods in scope ClassMethods as.. class methods!
        mod2 = mod.const_get("ClassMethods")
        extend mod2

        # If the newly included trait has a initialize_trait method in the ClassMethods-scope:
        # ... call it with the options provided with the trait-line.
        if mod2.method_defined?(:initialize_trait)
          initialize_trait(options)
        end
      end
    rescue
      puts "Error in 'trait #{trait}': " + $!.to_s
    end
  end
end
class << self; alias :has_trait :trait;  end
```

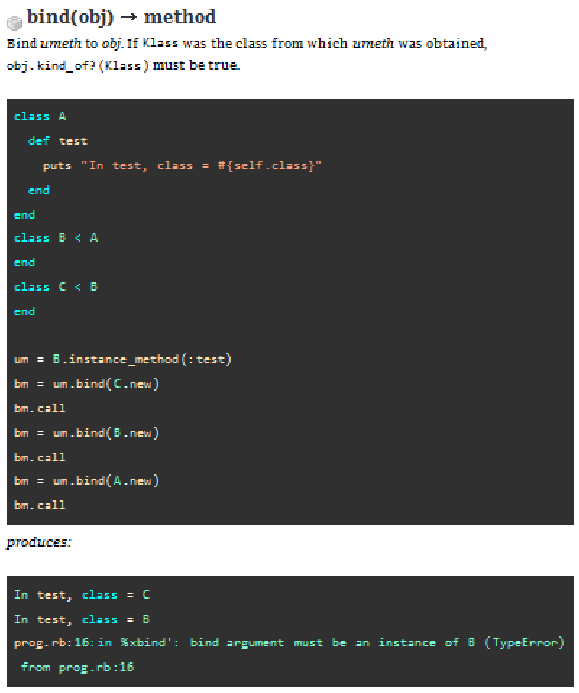
11.8 Method call benchmark

```ruby
module Traits
  module Hittable
    def update
      "update in hittable"
    end
  end

  class MonsterWithTraits
    include Traitable

    def update
      "update in monster"
    end

    incorporate.traits(:hittable)
    .resolve(:update)
      .with_lambda { update_in_monster_with_traits + update_in_hittable }
    .done

  end

  class MonsterWithMixins
    include Hittable
    def update
      "update in monster" + super
    end
  end
end


n = (1e7).to_i
Benchmark.bm(32) do |b|
  b.report('Call method within class with mixin: ') do
    monster = Traits::MonsterWithMixins.new
    n.times do
      monster.update
    end
  end
  b.report('Call method within class with trait: ') do
    monster = Traits::MonsterWithTraits.new
    n.times do
      monster.update
    end
  end
end
```

Output:

```
                                       user     system      total        real
Call method within class with mixin:   7.250000   0.000000   7.250000 (  7.310547)
Call method within class with trait:   8.641000   0.000000   8.641000 (  8.706054)
```

11.9 Code and documentation of the method bind



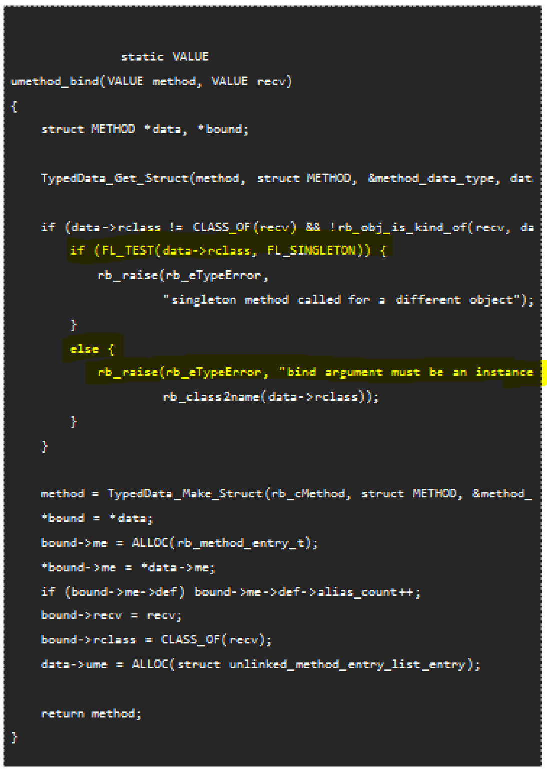
bind(obj) → method

Bind *umeth* to *obj*. If `Klass` was the class from which *umeth* was obtained, `obj.kind_of?(Klass)` must be true.

```
class A
  def test
    puts "In test, class = #{self.class}"
  end
end
class B < A
end
class C < B
end

um = B.instance_method(:test)
bm = um.bind(C.new)
bm.call
bm = um.bind(B.new)
bm.call
bm = um.bind(A.new)
bm.call
```

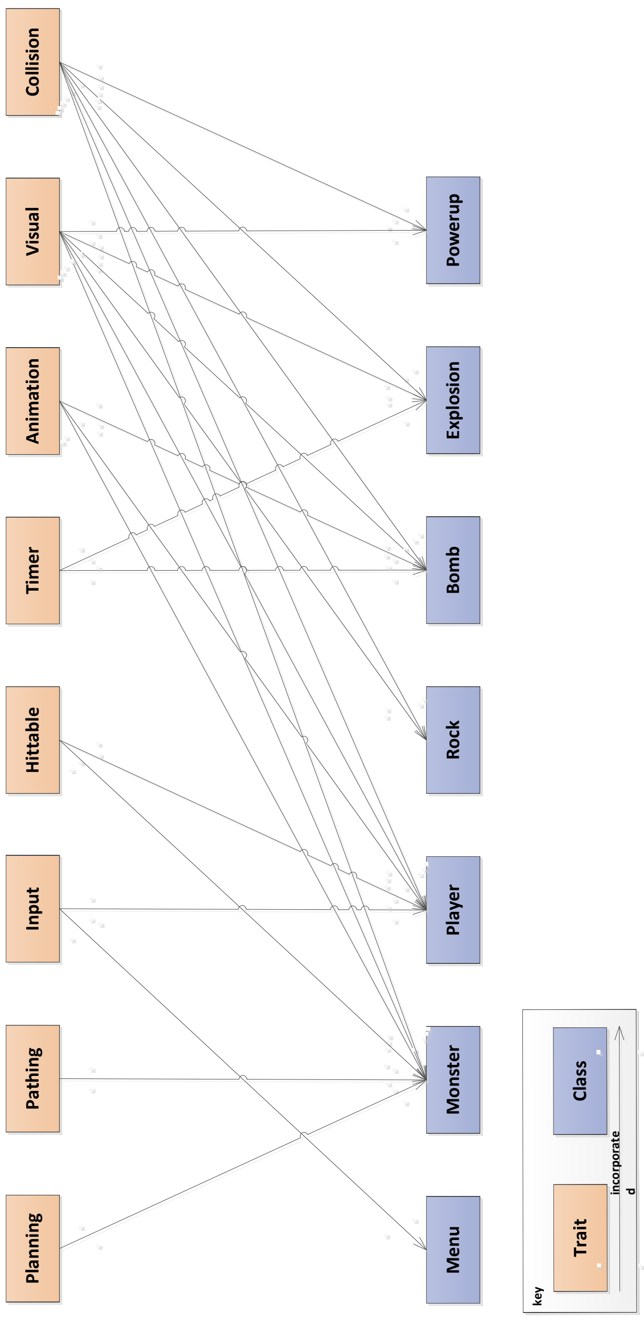
produces:

```
In test, class = C
In test, class = B
prog.rb:16:in %xbind': bind argument must be an instance of B (TypeError)
 from prog.rb:16
```



```
              static VALUE
umethod_bind(VALUE method, VALUE recv)
{
    struct METHOD *data, *bound;

    TypedData_Get_Struct(method, struct METHOD, &method_data_type, dat

    if (data->rclass != CLASS_OF(recv) && !rb_obj_is_kind_of(recv, da
        if (FL_TEST(data->rclass, FL_SINGLETON)) {
            rb_raise(rb_eTypeError,
                     "singleton method called for a different object");
        }
        else {
            rb_raise(rb_eTypeError, "bind argument must be an instance
                     rb_class2name(data->rclass));
        }
    }

    method = TypedData_Make_Struct(rb_cMethod, struct METHOD, &method_
    *bound = *data;
    bound->me = ALLOC(rb_method_entry_t);
    *bound->me = *data->me;
    if (bound->me->def) bound->me->def->alias_count++;
    bound->recv = recv;
    bound->rclass = CLASS_OF(recv);
    data->ume = ALLOC(struct unlinked_method_entry_list_entry);

    return method;
}
```

11.10 An example for traits without composition



Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den ________________ ____________________________