Services in Game Worlds:

A Semantic Approach to Improve Object Interaction

Jassin Kessing
July 29, 2009

TU Delft Delft University of Technology

Services in Game Worlds:

A Semantic Approach to Improve Object Interaction

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

MEDIA AND KNOWLEDGE ENGINEERING

by

Jassin Kessing
born in The Hague, the Netherlands

Computer Graphics and CAD/CAM Group
Department of Mediamatics
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Delft, the Netherlands
http://graphics.tudelft.nl/Game_Technology

Colophon

Personal information
Name: Jassin Kessing, BSc
Student number: 1217534
E-mail: jassinkessing@gmail.com

Thesis committee
Prof. dr. ir. F.W. Jansen
Dr. ir. R. Bidarra
Dr. ir. C.A.P.G. van der Mast
T. Tutenel, MSc

Cover picture
People Trading Services (2008)

Abstract

To increase a player's immersion in the game world, its objects should behave as one would reasonably expect. For this, it is now becoming increasingly clear that what game objects really miss is richer semantics, not eye-catching visuals. Current games' lack of semantics is mostly due to the complexity of designing semantic objects. If game designers would have to design all semantics by themselves, and game programmers would have to implement the handling of these semantics, game development time would increase enormously.

The goal of this research is to improve the semantics of game objects (or more properly, entities) in virtual worlds, resulting in more and better object interaction in games. This thesis proposes a solution in the form of *services*, describing interaction possibilities between entities, and a structure that makes it easier to specify them. An example of this is the service of a vending machine, which exchanges a coin supplied by a player for a soda. By introducing several components, such as classes, attributes and actions, a service is defined as the capacity of an entity to perform a particular action, possibly subject to some requirements. To incrementally specify and add services to game objects, a three-phased methodology is presented.

This approach has been implemented and validated by means of a prototype system, which enables a simple and intuitive definition of services in an integrated environment. By using two different editors, services can be defined during the game development process. In turn, a semantics engine is charged with all service handling during a game itself.

It is concluded that if game designers are presented with the tools to easily add semantics to game objects, these objects become aware of their services, therefore facilitating more and better object interaction, resulting in a much deeper gameplay experience than in current games.

Preface

This research would not have been possible without the help of others. First, I would like to thank the Faculty of EEMCS for giving me the possibility to do the research at the university, and by providing me a comfortable place. I also owe much gratitude to my friends and family for their continuous support. Furthermore, I am greatly indebted to the other graduate students, as the time spent together in the lab and the canteen was always very pleasant.

Above all, I would like to thank the people of the Game Technology group, in particular Rafael Bidarra and Tim Tutenel. Without the help of Rafael, I would not have been able to participate in this research for the GATE (Game research for Training and Entertainment) program. Furthermore, he assisted me by sharing his own vision about services, and giving useful tips and advice about the thesis. I would like to thank Tim for the discussions we had about the design of services, the recommendations for literature to consult, implementation details, and the useful comments about the thesis. I hereby wish him good luck with the continuation of his PhD work, hoping that my work will be put to good use there. The three of us also worked hard on a short paper about this research, and we are glad that it has been approved to appear in the proceedings of ICEC 2009.

Jassin Kessing
Delft, the Netherlands
July 29, 2009

Table of Contents

List of Figures and Tables

1. Introduction

Look around and you will probably see objects scattered all around the place. If the same room would be used as the virtual environment of a game, one would probably want to see the same objects – or at least some objects – because empty environments are unnatural to walk through. Game environments that are filled with objects will therefore help immerse the player into the game world. By using nice graphics, realistic animations, and real-world physics, virtual objects could appear as players expect.

Suppose you are standing in front of a table with a flashlight lying on it. Pushing it with your hand will probably make it fall down. When seeing a flashlight in a virtual room, one would expect that the same thing will happen. However, this was not always the case. In older games, objects were static and would stay on the same spot, whatever the player tried to do with them. Sometimes, objects could be picked up, but then they actually just disappeared from the scene and were added to the player's inventory.

Nowadays, game objects will behave more like real world objects. A flashlight in current games will fall down from the table when the player pushes it. This is done by using physics. A separate physics engine makes sure that objects behave accordingly when gravitational forces are applied. Furthermore, it prevents them from moving through other objects or the environment. So, when a box is pushed over an edge, it will spin in the air, accelerate towards the floor, crash, and roll a little further before stopping to move. However, depending on the velocity and the material of the box, the box can also break when hitting the floor. This physics, which makes use of the object's physical properties, is now starting to appear in games nowadays. With a physics or destruction engine, like *Havok Physics* and *Havok Destruction* (Havok, 2009), a weight is taken off the shoulders of game programmers, because they do not have to implement complex physics formulas anymore.

Let us go back to the real world. When picking up the flashlight, you can switch it on, and if it is powered by a battery, it will give you light. In game worlds, one would expect that the same will happen. However, even nowadays, that is mostly not the case, because most objects in games are still useless, being there for decoration purposes only. Only a few objects, which are crucial for the game progress, are given behavior. A few examples of games with functional objects will be given.

Probably the game with the most functional objects is *The Sims* (Maxis, 2000). In the game, shown in Figure 1, the player takes control over a family of Sims, who live together in a house that can be fully customized by the player. From a building mode, new objects, like furniture or kitchen materials, can be bought. Sims have needs/desires, which the player should fulfill in order to keep them happy. To do so, players command their Sims to interact with other Sims, gain experience in some domain to get

promoted at their jobs, or, in most of the cases, use their household objects (which may or may not help in getting experience). By watching television, for instance, a Sim's fun factor will rise. However, working out on a bench will lower his energy, but will increase his experience.

In many adventure and puzzle games, objects can be picked up and used on specific locations to trigger an event. This way, the player will, for instance, get access to a new area, or receive another object which should be used in another puzzle. An example can be found in the game *Myst* (Cyan Worlds, 1994), where the player should solve a puzzle in order to get a key, which can be used on a door to continue exploring.

In two other games, *Zack & Wiki: Quest for Barbaros' Treasure* (Capcom, 2007) and *The Incredible Machine* (Jeff Tunnell Productions, 1992), multiple objects can play a role in one big environmental puzzle. Only when several objects are placed on the correct spot, a chain reaction of events is triggered, which eventually reveals the treasure chest or ends the level.

Objects can also have an effect on the player's avatar (the personification of the player in the game world). In the role-playing game *The Elder Scrolls IV: Oblivion* (Bethesda Game Studios, 2006), bread can be picked up and eaten, which will raise the health of the avatar. A screenshot can be seen in Figure 2. Related to this is the armor that can be picked up and worn, which increases the level of defense. However, there is a difference between both objects. The apple will only give health once, while the armor will have effect as long as the avatar wears it.

One final example is the action-adventure game *Alone in the Dark* (Eden Studios, 2008). In the game, several objects, which look useless at first sight, can be combined to create a functional object: a full battery and an empty flashlight will provide a light in the dark when combined. Using objects with the environment can also modify their effect. A power cable can be used to shock the enemy to whom the cable is pressed against, but it will electrify all enemies that are standing in the water if the cable is put in the water.

Figure 1. A screenshot of *The Sims*

Figure 2. A screenshot of *The Elder Scrolls IV: Oblivion*

1.1 Problem Description

In all examples above, the behavior of each object (including its meaning, roles, etc.) was thought up by the game designer and implemented by the programmer. In the real world, a particular object may as well assume other functions or roles never anticipated by its designer; with a game object in a virtual world, this is definitely not (yet) the case, and certainly not automatically. This limitation makes it

impossible for a player to (make his avatar) interact with a game object in many reasonable ways. Although this is not a serious issue for games to be entertaining, it is very disappointing that current games still lack this common interaction functionality, because it reduces the freedom and creativity of the player. Suppose there is a puzzle game which makes use of objects, and the designer only thought of one solution. Now suppose that there is a player who comes up with another brilliant solution. If the player would not get the extra bit of freedom to use the objects the way he wants, just because the designer forgot to foresee this creativity, he would be rightly confused or disappointed.

We can describe this limitation more formally, saying there is a lack of object *semantics*. In the fields of linguistics, computer science and psychology, semantics is the study of meaning in communication. When focusing on virtual environments for computer games, semantics is the information conveying the meaning of (an object in) a virtual world (Tutenel, Bidarra, Smelik, & de Kraker, 2008). If game developers would add semantics to objects in games, this limitation would be resolved. However, this is not as easy as it seems, because it requires a lot of work to cover the semantics of all game objects. Furthermore, there is in current game development a serious lack of tools to easily specify and add this semantics to virtual objects.

1.2 Research Goals

The role of semantics in virtual environments is receiving increasing attention, but so far not much research has been done on adding semantics to game objects, let alone with the purpose of making them more functional or improving the overall gameplay. An exception to this is the PhD work of Tim Tutenel, who is doing research in the use of semantic information of objects in a virtual world. This MSc research is just a piece of that puzzle. With a semantically rich object representation, virtual objects assume behavior like in the real world, instead of consisting of a geometric model only. This can be illustrated with a few examples. When eaten by a character, an apple will reduce the hunger level of that character; in other words, an apple provides the *service* of satisfying someone's hunger. A coat serves its wearer for warmth. A fire, however, will provide warmth to everyone in the area. In these examples, objects provide a service. As can be noticed, the service can have an effect on just the user of the object, but also on the whole environment. Furthermore, the lifetime of effects can be different, because the apple can only be eaten once, while the coat will still exist after it is used. Some objects also require something, in order to provide their service. A vending machine is a good example, because it provides the service to supply cans of soda, but only after it has received money.

Empowered with the notion of services, objects acquire their own behavior, instead of a purely predefined behavior; they can behave as one expects, and correspondingly one is able to interact with them as one expects, regardless of whether the virtual object is completely imaginary or mimicking some real world object. This can significantly improve the gameplay, and make the game more entertaining, as players will be able to express their creativity and find more paths to achieve the same goal. For puzzle type games, for instance, the players can become much more creative in solving puzzles. For free-roaming games, multifunctional objects will immerse the player more into the game, because the world will be – and behave – more realistically. Even if the virtual world is not used for a game, but instead for professional training or therapy purposes, in which it is even more important to mimic reality, more functional objects would immerse the user more into the world as well.

This thesis focuses on our research efforts to achieve the following ultimate goals:

A. improve the semantics of objects within game worlds, by extending the behavior commonly found in games;
B. achieve more and better interaction between players and objects.

With these goals, we hope to give game developers the opportunity to expand the gameplay in interesting and profound ways.

1.3 Approach

In order to achieve the research goals, we propose to base our approach on the following steps:

1. analyze game objects in order to define a common structure to formally express them;
2. formally define the notion of services;
3. integrate the declaration of services into the object design process;
4. create the tools to design game objects with services;
5. handle the services of objects in games.

The first two steps in this approach are formalities, and are used to come up with definitions to avoid misconceptions later on. In particular, the notion of services is proposed, by which virtual objects get to 'know' about their roles in the world, how they can affect other entities (including the player's avatar or artificial agents), and how others can interact with them. For the third step, a three-phased methodology is presented to incrementally specify and add services to game objects. However, enabling game developer teams to do so requires the correct tools. An implementation and a validation of this fourth step is therefore presented by means of a prototype system, which enables a simple and intuitive definition of services in an integrated environment. This is a major step towards the first goal. The second goal can be achieved by successfully applying the fifth step, which introduces an engine to take care of all services of objects in the game world.

The thesis is structured as follows. First, related work about game objects, ontologies, and semantics is described in Chapter 2, in order to learn from their pros and cons, and find out which elements are useful in the design of services. Then, in Chapter 3, a general approach is presented about how semantics can be added to game objects. In Chapters 4 and 5, the components and services of this approach will be described in full detail. This is followed by implementation details in the next two chapters. Chapter 6 discusses the Library Editor and Level Editor subsystems of the prototype which were developed for game design purposes. For in-game purposes, the Semantics Engine has been created, which is described in Chapter 7. To demonstrate the potential of the approach and the use of the prototype, a Tech Demo and a few scenarios and use cases are presented in Chapters 8 and 9 respectively. Final conclusions, recommendations, and future work are discussed in Chapter 10. Appendix A gives some insight into the implementation details of the prototype. Finally, Appendix B contains the paper that has been accepted for the 8th edition of the International Conference on Entertainment Computing.

2. Related Work

This chapter will discuss previous research work about game objects and semantics. Although detailed information about the discussed topics can already be found in the Research Report of the author, the most important ideas will be summarized here.

First, Section 2.1 will discuss game object systems from the past and present. This is followed by ontologies and semantics in Sections 2.2 and 2.3. A successful approach for semantic objects, smart objects, is described in Section 2.4. Finally, Section 2.5 will present research about automatic scene composition with the help of semantics.

2.1 Object Systems

When games like *Space Invaders* (Taito Corporation, 1978) dominated the market, the number of game objects was pretty low and object attributes were simply hard-coded by programmers. When the object-oriented programming approach was introduced, objects in more complex games were categorized in structures (classes) with custom fields, like 'health' and 'attack power', next to the basic attributes like 'position' and 'orientation'. Game programmers created object systems, which were responsible for maintaining the data of objects. This resulted in an object design process that enabled game designers to specify objects, artists to create models for them, and level designers to add instances of those objects to the game world. These systems worked fine until games became bigger and therefore more complex. Objects started to become more specific, and more fields were required. Furthermore, to maintain and respect the distinction between designers and programmers when more dynamic game content is added, object systems had to be easy to use by designers, without the need to step into code or interfere with the programmers.

Instead of having data-driven object *properties*, the object *structure* can be data-driven as well. So far, systems have been object-centric, where the data belongs to objects. Figure 3a shows two objects, indicated by their IDs, with defined properties and their values. For people, this feels more intuitive and it will therefore be easier to create a new object. However, the game developers Scott Bilas (Bilas, 2002), Doug Church (Church, 2007), and Alex Duran (Duran, 2003) all pointed out on game conferences that it could be more useful to have a component-centric view. This way, each component is a self-contained piece of game logic, for which objects are just unique IDs. In a component, pairs of IDs and data are stored. So, in the component *Health*, pairs like "*ID0001: 90*" and "*ID0002: 60*" can be found, which means that the object with ID 0001 has a health of 90 (Figure 3b). When game designers want to create an object in an editor, they can then simply add components to objects and enter a value

for each component. Under the hood, however, the object ID is stored at the correct component. This means that the behavior of an object is determined by the data that is stored at the components. Working with a component-centric system will make the storage more efficient, and it will give a better overview of the components that can be associated with objects.

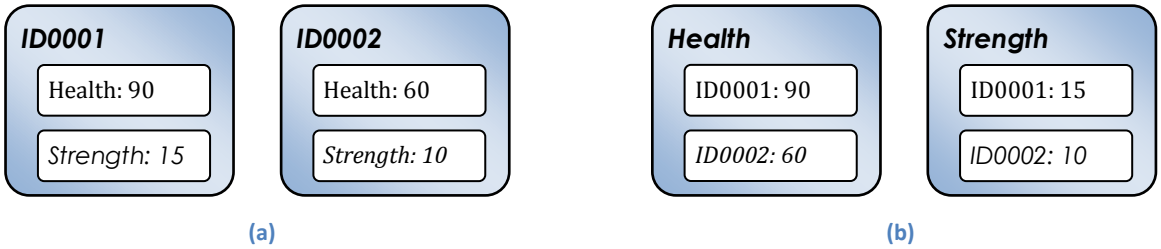

Figure 3. An object-centric vs. a component-centric object system

A combination of the object-centric and component-centric object system approaches is useful in the design of services, as it is possible to profit from the advantages of both systems. The component-centric approach is efficient and reusable for storage purposes, but it makes somewhat laborious to request which components belong to an object. The object-centric approach does not have this problem, because all object properties are stored in the object itself. The use of these approaches will become clear later on.

2.2 Ontologies

In the 80s, research in artificial intelligence (AI) proposed the notion of ontologies, due to the lack of shareable and reusable knowledge bases. An ontology is an explicit specification of a conceptualization: a representational vocabulary for a shared domain of knowledge, in the form of human-readable and machine-enforceable definitions of classes, relations, functions, and other objects (Gruber, 1993). In the context of computer and information science, it defines a set of representational primitives with which to model a domain of knowledge. Here, the primitives are classes, attributes, and relationships, which include information about their meaning and constraints. When placing ontologies in the context of this research, they define the meaning of objects and the relations between them.

Ontologies propose relationships among classes of meaning. Some relationships that can occur are *owns*, *causes*, and *contains.* The most important relationships, however, are the following (Huhns & Singh, 1997):

- *generalization and inheritance*, where classes and more refined versions of them are connected, and where each subclass inherits the features of its super class. The class *Car*, for example, has the class *Vehicle* as its parent;
- *aggregation*, where classes representing the components of something are associated with the class representing the entire assembly. An example is the relationship between *Wheel* and *Car*;
- *instantiation*, which relates a class with each of the individuals that constitutes it. A *Ferrari*, for instance, is a kind of *Car*.

In Chapter 6, the usefulness of these relations for the creation of objects with services will become apparent.

2.3 Semantics

When focusing on virtual environments of games and simulations, semantics is the information conveying the meaning of (objects in) a virtual world. A distinction can be made between three levels of virtual world semantics (Tutenel, Bidarra, Smelik, & de Kraker, 2008):

- the lowest level is the object level, where objects have physical and functional information. Although the physical information is often provided by designers of virtual worlds, functional information is not;
- on a higher level, the focus lies on the relationships between objects. Many relations can be found, but inheritance is the most important one, which will lead to several hierarchy possibilities. By creating an object hierarchy, related objects could be provided with the same functional information;
- the semantics on the highest level are global, and take aspects like time, weather and climate properties into account. With them, it is possible to create lush environments with vegetation, and objects that will be affected by the weather.

By providing this semantics to virtual environments, the amount of time, effort, and money spent in the process of creating these environments can be reduced, while making the world much richer with object behavior. The research described in this thesis will focus on the first two levels.

2.4 Smart Objects

Smart objects were a successful proposal for adding semantics to virtual objects (Kallmann & Thalmann, 1998) (Kallman, 2001). These objects were meant to deal with many of the possible user interactions in a virtual environment. For computer controlled agents, interacting with smart objects is much easier, because the required interaction information is stored in the object as predefined (scripted) plans. To be able to store information in an object, four different interaction features were designed:

- *intrinsic object properties* are similar to design properties like physical properties and a movement description of moving object parts;
- *interaction information* is used so that each agent will know how to interact with an object. This includes information about which parts of the object are usable, which shape the user should make with his hand, or where he should stand (relative to the object);
- *object behavior* defines what an object can and cannot do, depending on its state variables;
- *agent behavior* is the expected behavior of the user, which is useful to guide the agent. This way, he knows, for example, where to put his hand when using the object.

Noticeably, smart objects were primarily devised for manipulation, animation, and planning purposes, like grasping, pulling, or rotating (individual parts of) objects. An example is an artificial agent that can open a door by moving its hand to the door knob, using the correct hand posture, and turning the knob. Although smart objects are powerful for these purposes, they lack the information of which services

they provide to their users. However, their reusability makes them very useful: created smart objects or interaction features can easily be reused for the creation of new objects. Still, because of the current disadvantages of smart objects (creating behavior scripts for each object is a complex and tedious task for the designer) they have not been applied to commercial virtual worlds or games. Object services can therefore be a useful extension to this model, as a separate feature, to decrease the complexity of creating objects.

To model the interaction steps in agent-object interaction, an extended framework has been presented by Peters, Dobbyn, MacNamee, and O'Sullivan (2003), in which *user slots* can be associated with smart objects. User slots are based on the interaction features, but avoid concurrency problems because they do not make use of fixed rules. An agent can only use an object when he has obtained a free user slot for it. A disadvantage of the slots is that each time an agent wants to use the same object, the same interaction behavior will occur. To overcome this problem, several user slots and more animations are provided. To use the object, the agent can consult the *usage steps* that are stored with the user slot, which, in turn, contain the following information:

- the agent's movement animations;
- conditions to be met to continue with the next step;
- object or user attributes that will be changed;
- whether it is possible for the user to interact with other agents;
- positions on the object upon which the agent should focus (for gazing purposes);
- any remaining information.

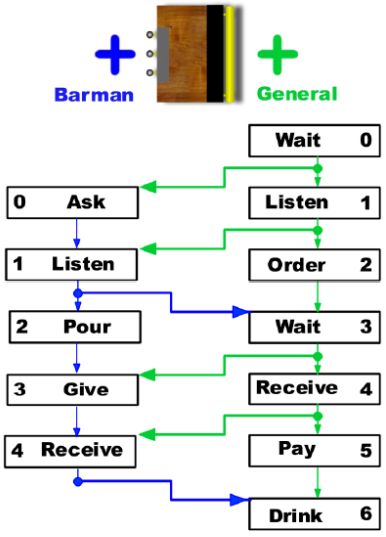

Figure 4. A smart Bar object with user slots, usage steps, and conditions

In Figure 4, an interaction diagram can be found of a smart object which represents a bar scenario. An intelligent agent is placed at the user slot called *Barman*. Another agent, the customer, can make use of the user slot *General*. In order for the customer to receive a drink, he should follow the usage steps that are shown. In the figure, usage steps are represented as boxes connected with arrows that represent conditions for the next step. At each step, in which the information as mentioned above is stored, the

movement animations of the avatars change and it is possible that several attributes change as well, like the amount of money that the user has in his purse when he pays.

2.5 Scene Composition

In the creation of games and simulations, the virtual environments are created and filled with objects afterwards. This can be done by hand by the level designers, but it would be a lot easier if objects could be placed automatically in the environment. However, to make sure that those objects are placed at the correct (logical) positions, semantic connections between objects are needed.

In the IConS modeling system, relations between objects are described as surface constraints, where each object has predefined offer and binding areas (Gösele & Stuerzlinger, 1999). An offer area is a place on an object where other objects can connect to, and a binding area indicates the place of an object with which it can connect to other objects. An example is a lamp, of which the bottom can be placed on the top of a table. In further research, the IConS system has been extended with dual constraints (Smith, Salzman, & Stuerzlinger, 2001). In the original system, only parent-child relationships could be created. A cabinet could only be placed against the wall, but not against another cabinet. The dual constraint in this new system, which has been called MIVE, prevents this restriction.

In related research, surface constraints indicate how objects can be placed on surfaces of other objects, proximity constraints indicate how close objects should be to others, and support constraints are used to indicate whether an object can support other objects, or the other way around (Xu, Stewart, & Fiume, 2002). A difference with the MIVE system is that all objects are classified, which results into a semantic database. This way, for example, all kinds of tables (coffee table, dining table, etc.) will know that they can support other (classes of) objects, like lamps. Now, the designers do not have to define all constraints over and over again, but can just assign them to the correct object class.

The object association that is used in the MIVE system is very powerful. The idea could be useful in combination with services. For example, before a vending machine can offer its service (providing a soda) to the user, it should receive electricity. The service of a power cable is to supply electricity. If the cable knows at which place (a binding area) he can supply the electricity, it can be connected to the (offer) area of the vending machine where it should get the electricity. By making use of services and constraints, it should be possible to automatically compose scenes, and fill them with objects that should logically be connected to each other. More research towards the use of semantics in the design of virtual worlds has been done by Tutenel et al. (2009)

In the remainder of this thesis, it will become clear where the related work of this chapter is on the interface with the current research.

3. A General Approach to Add Semantics to Game Objects

Chapter 1 made clear that game objects miss richer semantics. Although present in some virtual environments, semantics is not yet available in game objects in order to improve the gameplay. This is mostly due to the complexity of designing semantic objects. The research discussed in this thesis focuses on a solution for this problem in the form of an approach to add semantics to game objects. This is done by introducing services. A brief overview of this approach is discussed in this chapter, with more details in the next chapters.

This chapter is structured as follows. First, in Section 3.1, game objects will be analyzed in order to discover the components of which they consist. Then, Section 3.2 will introduce the notion of services. Finally, Section 3.3 will present an approach to declare services in several phases.

3.1 Subdividing Game Objects into Basic Components

In order to design services for game objects, it can be very useful to analyze how real world objects are structured. After all, game objects are mostly based on real world objects. First of all, one remark should be made. When discussing real world objects, the word 'physical entity' is more appropriate than the word 'object'. The reason for this is simple: the term 'physical entity' is more general, and is applicable to more than real world 'objects' only. Consider a substance like water or a physical phenomenon like electricity. None of them are objects, but both of them are physical entities with several services. The word 'entity', instead of 'physical entity', on the other hand, would be too broad, because abstract emotions, like 'sadness' and 'love', are entities as well, considering their descriptions in dictionaries. For simplicity, this thesis will interchangeably use the words 'object' and 'entity', but one should keep in mind that, unless defined otherwise, the word 'physical entity' will be meant.

Each object in the real world can be said to belong to some *class*, which is in the field of object-oriented programming defined as 'a group of objects which have similar characteristics and exhibit similar behavior' (Barnes, 2000). A 'similar characteristic' will be formalized by the notion of *attribute*. It is this notion that makes a difference between classes, just like the object-centric approach of Section 2.1. The exhibition of 'similar behavior' is an informal way to describe a *service*, but this is further described in the next section. To formalize the notions of class and attribute in this context, they can be defined as follows:

- *Class*: a generic description of a collection of physical entities based on their essential common attributes

- *Attribute*: an inherent characteristic of a class

Classes describe entities, varying from physical objects like *couch* and *human*, to substances like *water*. For attributes, one can think of abstract attributes like *comfort*, or physical attributes that are related to the material of an object, like *mass* or *friction coefficient*. One might wonder why the term *attribute* is used, instead of *property*. According to WordNet (2009), a *property* is 'a basic or essential attribute shared by all members of a class'. This shows that a property is a kind of attribute. By using attributes instead of properties, it is possible to also include feelings and emotions, which are attributes, but not properties. Feelings and emotions might then be useful to assign to the classes of living entities, like humans.

Having only attributes is insufficient, as there should also be a way to represent them, for example with values. For an attribute like *comfort*, a quantified range (e.g. 0 – 100) between *uncomfortable* and *comfortable* will do. For other attributes, however, this will not suffice. Therefore, the components *unit* and *state* are introduced:

- *Unit*: a standard quantity in terms of which other quantities may be expressed

- *State*: the condition of an entity at a particular time

On the one hand, attribute values could be expressed as quantities with a particular unit, while on the other hand, they could adopt a particular state. The *mass* attribute could be expressed with a numerical value in combination with the unit *kilograms* or *ounces*, while *edibility* could be expressed in the state *edible* or *inedible*. To simplify the assignment of attributes to classes, the *material* component is introduced:

- *Material*: the matter of which an entity is made

A material brings several material attributes along, to indicate whether the material is *waterproof* or *fire-resistant*, for example. When a class is related to a particular material that is waterproof, the entity that is represented by the class will be waterproof as well.

Chapter 4 will describe each of the discussed components in much more detail.

3.2 Designing Services

With the notions of *class* and *attribute* (and the subordinate notions of *unit* and *state*), entities can be identified and distinguished from each other. The object-oriented definition of a class defined similar

behavior for objects in a class. This behavior forms a foundation for the definition of a service. In the real world, entities provide particular services, which should also be the case for entities in a virtual world (that is, if we want to mimic the real world). In order to formalize the notion of services, it is useful to study some simple examples.

One example is the service of a vending machine. When a person inserts the right amount of coins, the vending machine will supply one of the sodas from its inventory. This resembles the economic model of *supply and demand*, which describes 'the effects on price and quantity in a market' (Wikipedia, Supply and demand, 2009). Although the economics in this model are irrelevant for this research, the notion that quantity is demanded by consumers and quantity is supplied by producers seems to be very useful. The service of a saw machine also matches this notion of supply and demand. When the machine is given trunks (it demands trunks), it can process them into wooden planks (it supplies planks). When looking carefully at both examples, one might notice that there is a difference between the supplies, which is something that should be kept in mind. In the first case, the supply was given from the inventory, while in the latter case, a transformation process occurred.

The previous notion of supply and demand is unfortunately not sufficient for services of every class. Consider a coat, of which one service is to provide warmth, but only when it is worn. In this case, there are no physical demands and supplies. Instead, this resembles an *action-effect* pair, where the action is wearing the coat, and the effect is receiving warmth (at least, from the user's perspective). Noticeably in this effect is the warmth, which is related to the temperature of the user: receiving warmth will make the temperature of the user rise. When using the terminology from the previous section, one could say that *temperature* is an attribute, the *human* class has (or: possesses) this attribute, and that the value of the attribute is changed. In this example, only the attribute of the user was affected, which is not always the case, as the next example shows. Consider an avatar who punches an enemy. The effect of this action is not only that the enemy's *health* is lowered, but also that the avatar's *fatigue* increases. Here, attributes of both entity classes were affected. For demonstration purposes, the effects were limited to these two, but note that others exists as well.

This latest example includes actions of entities in the definition of a service. To increase the complexity of formalizing the notion of service even more, four more notes can be made. First, it is out of the question that an action always has successful effects: an avatar can punch whenever he wants, but if there is no target, there will be no effect of lowering *health*; a *space heater* can heat the surrounding area, but if there is nothing to heat, there will be no *temperature* attribute to increase. A complete supply and demand pair is not required as well, as there may be entities which give away their supplies for free. A third note is about attributes: besides being affected by actions, there is also the possibility that attribute constraints are required for an entity to provide a particular service. To illustrate this, the punching example can be applied, as the *fatigue* of the avatar may require that it is not too high before being able to fight. The earlier given example of the vending machine can also be borrowed. Only when the device is powered on (in the state *enabled*, for instance), will it function. This example leads to the final note, as the usage steps of Section 2.4 might be needed here. Consider that the machine should be powered on, and a button should be pressed before it supplies a soda. One should then indicate that this should happen in this order, and not the other way around.

Although the previous examples all looked different, common service elements can be deduced from them. On one side, a service could have some *requirements*, like the demands. On the other side are *actions*, like the supplies. Formally, this research defines services, actions, and requirements as follows:

- *Service*: the capacity of an entity to perform an action, possibly subject to some requirements
- *Action:* a process performed by an entity, yielding some attribute value changes or (new) entities
- *Requirement*: an action or attribute constraint

When looking back to the given examples, it is possible to observe that each example is subject to these definitions. (This is also the case for demands and supplies, when considering them as actions: for a vending machine, the action *get coin* is a requirement, while it performs the action *supply soda* afterwards.)

The general idea that is proposed here is to define services for classes. In contrast to classes, attributes, materials, units, and states, services are not considered as basic components. The reason for this is that a service is too specific and in most cases dependent on a particular class. Consider a space heater, a coat, and a camp fire. They all offer the service of providing warmth, but in each case the requirements are different. Actually, it is not the service that is the same, but the action. Therefore, actions are considered as components instead, which makes a service an abstract concept. To combine the components with services, one could say that for each class (component), one or more services can be defined, having at least one action (component), and possibly having requirements in the form of an action (component), or an attribute (component) with a particular value, which is either a numerical value with a possible unit (component), or a state (component), where that attribute could be derived from the material (component) of that class.

Chapter 4 will give more details about action components, followed by detailed information about services in Chapter 5.

3.3 Services Put to Work

The concepts that have been developed in the previous sections are useful to define object semantics. This section will discuss how services can be practically used. Subsection 3.3.1 will discuss how they can be specified, while Subsection 3.3.2 will discuss their use in a game.

3.3.1 A Three-Phased Approach

So far, it has been unclear where services fit in the regular design process of game objects. Section 2.1 mentioned a process that consists of two phases: one phase where game objects are (visually) designed and (logically) specified and one phase where instances of those objects are placed inside the game world. If services are added to this design process, it feels natural to declare them in the first phase.

Although this would not be incorrect, we pose that an improvement to the object design process can be made first. Instead of two phases, three main phases can be identified, which are partly based on the regular design process. Although each phase is distinct, the order should be maintained, because the third phase needs the second one, which, in turn, needs the first one. With this three-phased design methodology, designing semantic objects is done step by step, and services are specified with an increasing amount of detail in each step. The identified phases are the following:

(i) a *specification phase*, in which generic classes are specified;

(ii) a *customization phase*, where a selection of classes is customized into concrete game-specific classes;

(iii) an *instantiation phase*, where object instances of these game-specific classes are placed in a game world.

Figure 5 gives an overview of these three phases. As can be seen in the figure, the regular object design process is left intact. The last two phases are part of a specific game development project, shown in the big box on the right. This process has been extended with an individual specification phase in the left-most box.

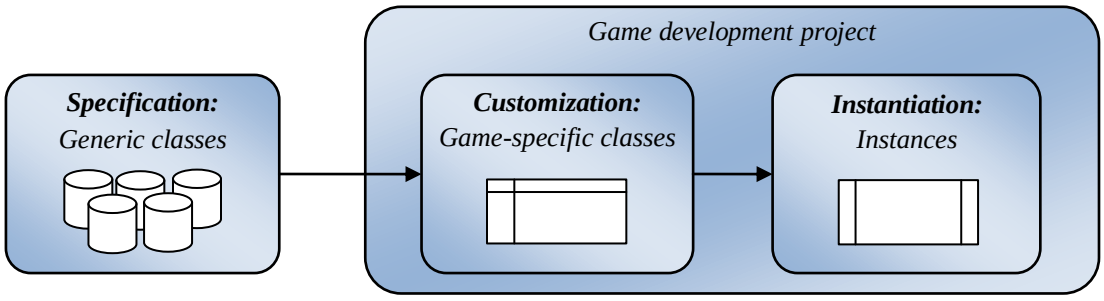

Figure 5. A three-phased design methodology for game objects with services

For each phase, libraries play a central role. These libraries are based on the components that were discussed in the previous two sections, meaning that each component type gets its own library. For the second and third phase, two new component types (and therefore two more libraries) will be introduced. In Section 2.1, component-based object systems were discussed, in which the components are self-contained pieces of game logic. The components of this chapter are based on them. Components are the building blocks for game objects, and their combined information gives meaning to them. Libraries are a means to maintain and organize all available components, and each phase is meant to establish relations among these components. To clarify this, each phase will be discussed in more detail below.

The key idea of the specification phase is to create a library of *generic semantic classes*. Many games take place in worlds that resemble each other, often because they are based on the real world. Although

the worlds may differ from each other because of the graphics, the semantics of their objects (when present) is often the same. Consider a chair: although present in many games, its behavior is redefined for each game, over and over again, which is a waste of time. The specification phase avoids this by introducing generic semantic classes, which are classes that only have to be defined once, after which they can be used in all kinds of virtual worlds. This stimulates consistency and reusability, and thus reduces game development time. In the specification phase, one could specify a generic *chair* class, and define that it is possible to sit on it.

Besides a library of generic classes, libraries of attributes, units, states, materials, and actions are also defined in this phase. By establishing relations among these components, semantics is specified. The *kilogram* unit, for example, could be related to the *mass* attribute, which could, in turn, be related to the *chair* class. The specification phase is also the right moment to define generic services for each class, in order to specify its semantics even further.

In contrast with the specification phase, the other two phases are not generic. Instead, they are specific for each particular game development project and comparable to the two common object design phases. After all, each game has its own unique environment, object style and desired behavior, etc. In these phases, where two more components are introduced, customization and instantiation play a central role. The customization phase is all about *game-specific classes* (which are now and then shortened to *game classes* in the running text):

- *Game-specific class*: a class that represents an entity in a specific game, derived from a generic class

An example of a generic class is a *book*, while a *book* with the title 'Semantics for Dummies' is an example of a game-specific class. By deriving game-specific classes from generic classes, all their generic semantics is automatically inherited, including their set of attributes, and the services they provide. This is a major advantage, as this generic behavior does not have to be defined anymore, which reduces the time to create these components. In a particular game project, it suffices then to customize the specific behavior desired for each game class, including its specific attribute values. The customization phase is also the time to assign the project-specific 3D models to the relevant game classes, and to indicate which animations of those models should be played when a particular action is performed. In this phase, new attributes can be associated with specific classes as well, just as the declaration of new services, in order to add even more specific behavior. The result of this phase is a library that is filled with game-specific classes. Finally, in the instantiation phase, *instances* of the customized game-specific classes can be created and placed inside a game world.

- *Instance*: a single occurrence of a game-specific class

For example, a game could contain multiple copies of the aforementioned Dummies book. If desired, the instance components can be customized even further, in order to create specific distinct instances. An example is a *key* that only unlocks one specific *door* instance, and not all other *doors* in the

environment. During this phase, an instance library is filled with all the instances of a game, including their customized behavior.

In a game development environment, it is the task of game designers to take care of the customization phase. After all, they are the ones to design the game world and its objects, and to define their behavior. 2D and/or 3D artists will aid them to create visual representations of those objects. The instantiation phase is intended for level designers, who create the actual game world(s) and place object instances at specific positions. It remains an interesting question when and by whom the generic libraries have to be created and when and by whom the components have to be related. Although this is not the topic of the current research, three possibilities are mentioned here. The least practical would be to let one individual design the libraries and relate their components, because this would be too much work, especially if we want to include each possible class, attribute, unit, etc. from the real world. A more obvious choice would be to let game designers incrementally fill the generic libraries during several development projects. In each subsequent project, there are more generic classes to be reused, while new relations can be established for some less-used classes. To extend this suggestion, the libraries could be publicly accessible for game designers all over the world, enabling everyone to create new relations. This collaborative approach, however, should be investigated much more carefully, as it deserves a research project on its own.

3.3.2 An In-Game Engine

Even when services can be specified in several phases, they are still useless if a game does nothing with this additional information. One should check whether the requirements of a service have been satisfied. Furthermore, entities should be made aware that they are then able to perform an action. And when this finally happens, the effects of that action should be taken care of. It is possible to assign all these complex tasks to game programmers, but this would give them much more work, which is obviously undesired. Instead, it is better to look at game techniques and find out how complex tasks are handled there. In game development, the driving factors behind complex operations are game engines, middleware systems that form a layer between game implementation and hardware operations. A rendering engine, sound engine, and a physics engine are the most common ones, but many more engine types exist. (If interested, one might start looking on Wikipedia. (2009)) With a physics engine, for example, game programmers do not have to implement the rules of physics, but should just indicate the mass and material properties of an object. Then, objects are passed to the engine, which takes care of falling objects, rotating objects, sliding objects, and so on. Each time an engine is updated, which is usually done when a game is updated, the engine calculates the new game state for which it has been specialized. Furthermore, most engines provide some useful functions for game programmers, which they can use to improve their game.

A specific engine to take care of services is very useful. Created instances can then be passed to this engine, after which it checks whether they have satisfied requirements, performs their actions if that is the case, and makes sure that other instances are affected by those actions. Furthermore, the engine could support the interaction between entities with services, which is schematically shown in Figure 6. Suppose that there is something that entity A (which could be a player) wants to have. He might then visit other entities and ask whether they can supply this ‘something’ (1st picture), a request

that could be taken care of by an engine. In case an entity B is able to do so (2nd picture), he can tell A his demands. If A is willing to give B his demands, an exchange takes place (3rd picture), controlled by the engine. In the end, both entities will be happy (4th picture).

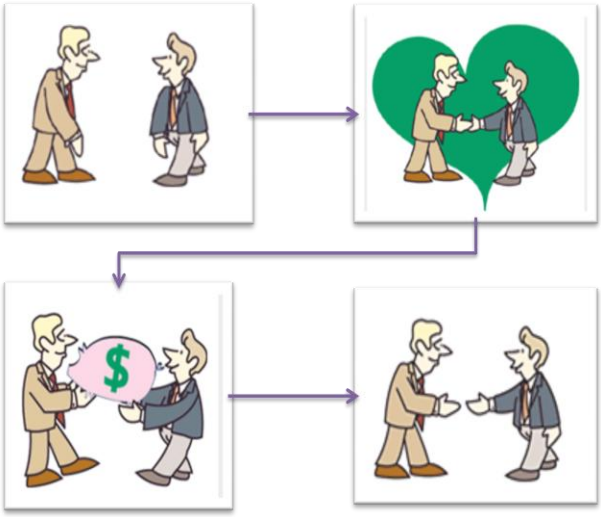

Figure 6: Interaction between two entities

The general approach that has been presented in this chapter will be elaborated in the next chapters, starting with the components and the services in Chapters 4 and 5 respectively. Chapter 6 describes how the three-phased design methodology has been implemented in a prototype. Then, Chapter 7 will discuss the implementation of the engine.

4. A Detailed Description of Components

In Chapter 3, the basic components (class, attribute, material, unit, and state) were first introduced. After formalizing services, the components were extended with actions. The phased approach finally presented two more components: game-specific classes and instances. This chapter will give much more details about all those components.

The order in which the components will be discussed will be a bit different from the order in which they were introduced. As attributes are used in several other components, they will be discussed first in Section 4.1. Because units and states are strongly related to attributes, they will be included in that section as well. Section 4.2 will then describe materials, followed by actions in Section 4.3. After this, it will be possible to describe generic classes, game-specific classes, and instances in Sections 4.4, 4.5, and 4.6 respectively, following the same order presented in the previous chapter. Figure 7 gives an overview of all the components, which are all stored in their own libraries. Note that an additional component library is present in the figure. It is a helper component that will be discussed alongside units.

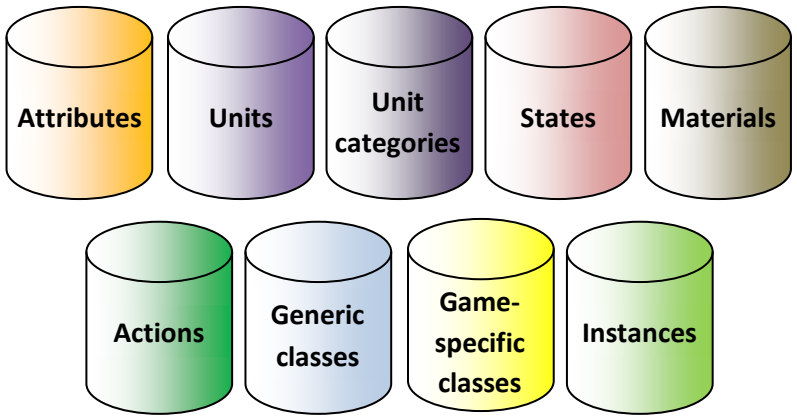

Figure 7. The component libraries

4.1 Attributes

Attributes have been defined as characteristics of a class. Although attributes distinguish one class from another, the attribute component itself is not that complex and interesting. One might say that an attribute needs a value, but this is only partly true. Attributes do need a value when they have been

associated with a generic class, game-specific class, or instance, or when they indicate the effect of an action. The attribute component, however, should not be aware of any value, as that value could be different for each use, and is therefore not related to the component.

What is common, are the units or states that an attribute can adopt. Units indicate the dimension of an attribute value when it is expressed as a real value: an integer or a floating point number. A value can, on the other hand, also be expressed as a state. Note that it is not required for an attribute value to be a state or to have a unit. After all, a value can be dimensionless. An example is the *friction coefficient*, which is used in dynamics and mechanics, and is expressed as a real value without a unit. An abstract example can be seen in Figure 8, where game class G has attribute A with real value *x*.

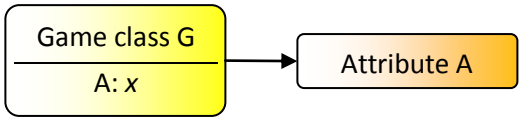

Figure 8. A game class with an attribute

4.1.1 Units and Unit Categories

To express attribute values with a dimension, *units* are required. If an attribute like *mass* is associated with a particular class, one does not only wants to know the value of that *mass*, but also the unit. After all, *mass* expressed in *kilograms* will be very different from *mass* expressed in *milligrams* or *pounds*. However, it should not be possible to express *mass* in *meters* or *degrees*. To make a distinction between these units, the International System of Units (abbreviated as SI, from Système International) was adopted in 1960 for the recommended practical system of units of measurement (Bureau International des Poids et Mesures, 2008). In this system, seven well-defined base units were chosen, which are regarded as dimensionally independent: the *meter* (unit of length), the *kilogram* (mass), the *second* (time), the *ampere* (electric current), the *kelvin* (thermodynamic temperature), the *mole* (amount of substance), and the *candela* (luminous intensity). By making combinations of base units according to algebraic relations linking the corresponding quantities, derived units are formed. Examples are the *area* (square meter), and the *luminance* (candela per square meter). Finally, prefixes were chosen for decimal multiples and submultiples, ranging from 10^{-24} to 10^{24}, to define, for example, *millimeters* or *centimeters*.

The unit components are heavily based on these standardized base units and their derived units. However, the unit library also contains some units that cannot be derived from one of the seven SI base units, but are very useful for the design of game objects. These are monetary units, like the *euro*, and computer memory units, like the *bit*. For units, a new component is introduced, which serves as a helper component:

- *Unit category*: a collection of units sharing the same base unit

Each unit category contains several units, of which one is selected as the base unit. So instead of assigning individual unit components to an attribute, a unit category has to be chosen, after which a

particular unit from that category can be chosen to express the attribute value in. This is shown in Figure 9, where game class G has attribute A with real value *x*, expressed in unit U of unit category C.

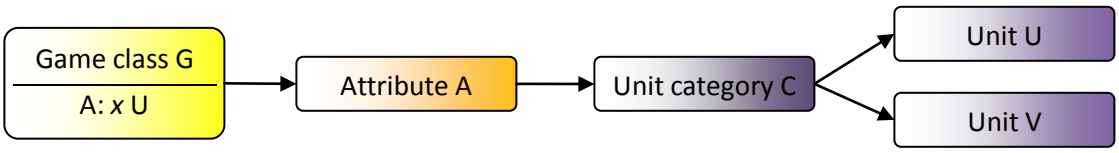

Figure 9. A game class with an attribute, expressed as a numerical value with a unit

As unit categories are aware of prefixes, the desired prefix can be selected as well. For this, one should keep in mind that prefixes are different for units of *time* and *computer memory*, but this should not be a serious bottleneck. Furthermore, each unit in a category should be aware of the equation to convert that unit to the base unit, and an equation to convert the base unit back to that unit. This way, game designers can choose their preferred unit when defining an attribute, while calculations with the values of those attributes later on can be done with the base unit (assuming the attributes have the same unit category, otherwise this is not possible at all).

4.1.2 States

In this research, *states* are used to express attribute values, in case they are not expressed with numbers. Although an attribute associated with an object can only adopt one state at a time (just like there can only be one numerical value at a time), there could be multiple states that an attribute can adopt over time. For example, a door could be opened now, and closed in ten minutes. In this case, the *opened* state switches to *closed*. State components are said to have a relation with at least one other state component; otherwise, an attribute will only be able to adopt one state. The *opened* state, for instance, will be related to its opposite: the *closed* state. Other example state sets are *on/stand-by/off*, *locked/unlocked*, *low/medium/high*, and *edible/inedible*. Three abstract related states are shown in Figure 10.

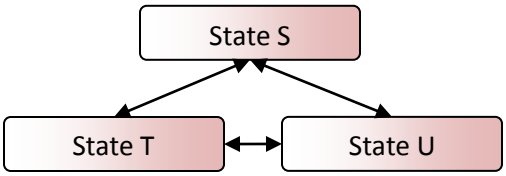

Figure 10. Three related states

One might argue whether states should be associated with attributes, or directly with classes. Both have a disadvantage. In the first case, it might be hard to find (the name of) an attribute that covers several states. Thinking of a perfectly suitable attribute for the *opened* and *closed* states is harder than it seems. In the latter case, it is questionable how totally unrelated states should be distinguished from each other. When states like *locked*, *unlocked*, *opened*, and *closed* are all associated with a door class, and classes can only adopt one state at once, there is no way to design an opened and unlocked door. Somehow, sets of related states should be kept together. As attributes are characteristics of an entity, and states are a way to express those characteristics, it is still the best solution to keep the states with attributes, even when it is hard to find an attribute to cover them. In this case, it is still possible to define a new attribute, specialized for a set of states. Otherwise, it is also possible to define the two Boolean states *true* and *false*, and attach those to an attribute.

4.2 Materials

Materials are matters of which an entity is made. Examples are natural materials like wood, fabrics like plastic, or textiles like wool. Materials have their own properties, as some are water-resistant, some are fireproof, and others are both. The melting temperature for each material is different as well. In this research, a material is a helper component that forms an optional bridge between classes and material-related attributes. Materials can be associated with attributes with particular values, after which they can be associated with classes, which will then inherit those attributes. A major advantage of this approach is that material attributes do not have to be defined twice for objects with the same material. For example, instead of specifying the same attributes twice for a *wooden bench* and a *wooden table*, the attributes are associated once with the *wood* material, which is then applied to the *bench* and the *table*. Figure 11 shows an abstract example of a class that inherits the attributes and their values of its two materials.

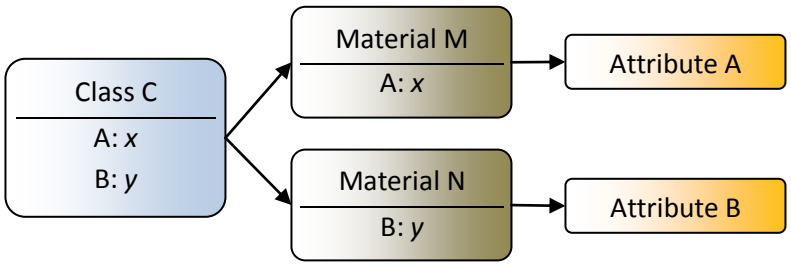

Figure 11. A class that inherits the attributes of two materials

A problem arises when multiple materials with contradicting attributes are applied to a class or game object. This, however, will then be a design fault. Section 4.4 proposes a solution to prevent this from happening.

4.3 Actions

In the definition of a service, an action can be found at two sides: one side states that a service is the capacity of an entity to perform an action, while the other states that an action can be a possible requirement for this performance. An *action* itself was described as 'a process that is performed by an entity, yielding some attribute value changes or (new) entities'. While the service-specific behavior of an action will be explained in Chapter 5, its common component behavior will be explained here.

Each process that an entity can perform is an action, which means that there are many possible actions, such as *walk*, *eat*, *play*, *break*, *kick*, *close*, and so on. Of all these actions, it should be obvious that a *human* will be able to perform many of them, in contrast with inanimate objects like *tables*. When examining these action examples, one might notice that there is a difference between the first three and the last three. *Walk*, *eat*, and *play* are actions that can last for an indefinite amount of time, while *break*, *kick*, and *close* are actions that are performed only once, starting at a certain moment, and ending some period afterwards. Of course, one is able to kick twice, but there is a significant difference between a *kick* action and a *walk* action, as the first is *discrete*, and the latter is *continuous*. This is an important notion for the effects of an action, and is discussed later on.

First, it is useful to state that an action is meaningless without an *effect*. After all, if there is no effect, nothing changes, as if the action has never been performed. However, this is independent from the fact whether the action was successful or not. In case of *success*, the action will have an effect, while in case of *fail* this might not happen. Consider the punching example that was given in Section 3.2. When a *person* punches an *enemy*, the *person* will always get a little *tired*, even if he missed the *enemy*. Only when he actually hits the *enemy* (the action is successful), the *enemy*'s *health* lowers. This example also shows two important elements of an effect. One of them is the entity that is affected, which is either the entity that performs the action, also called the *actor*, or the entity on which the action is performed, the so-called *target*.

Another element is that effects modify attribute values. Dependent on the value of an attribute, an effect either changes the state to one of the states that the attribute can adopt, or modifies the numerical value. Of course, if the affected attribute is already in the state that it should change to, the state remains the same. An example of a state change can be found at the *unlock* action: when a *person* performs this action on a *door*, independent of whether it is locked or already unlocked, the *locked*/*unlocked* attribute of the *door* changes to *unlocked*. Numerical values can be modified by increasing or decreasing them. Although an action component itself does not know with which quantities the attribute should be modified, as this is dependent on the entity that performs the action, it does know whether the value should be increased or decreased. The *heat* action, for example, will always increase the *temperature* attribute, while the *drink* action will always decrease the *level of thirst*. Figure 12 shows an abstract example of an action with two effects, which are related to attributes B and C. The first effect increases the value of B, while the second one changes the state of C (which can be expressed in the related states S and T) to T.

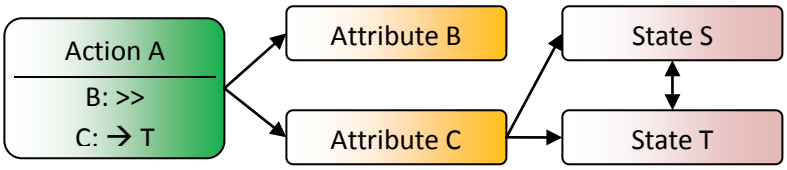

Figure 12. An action with two effects

For actions that are continuous, such as the *heat* one, it should be made clear how the value should be adjusted over time, a problem that is not available when having discrete actions. This, however, is again dependent on the entity with the service of performing that action, so this will be discussed in the next chapter about services. Furthermore, it should be noted that not all actions are directly executable by an entity, as is the case for the *get tired* and *die* actions, for instance. These type of actions are more like processes that can be triggered at some moment (when an entity runs out of health, for example). Therefore, for actions should be specified whether they are *executable* or *non-executable*.

A final aspect that should be mentioned when discussing actions is that some actions could mean the end of existence of an entity. An action could *exhaust* the target, as is the case for the *eat* action: when a *person* eats a *cupcake*, nothing of the cake will be left afterwards. On the other hand, an action could also destroy the actor, which is literally the case for the *explode* action of a *bomb*. Actually, the cupcake and the bomb are not entirely exhausted, but broken into tiny pieces. This research, however, does not deal with these situations, as it will only increase the complexity of the design and implementation, without achieving more useful interaction possibilities.

4.4 Generic classes

In this research, the *class* is the most important component, defined as 'a generic description of a collection of physical entities based on their essential common attributes'. Generic classes are the foundation of the presented approach, and are the key components in the first object design phase.

The attributes that were described earlier in this chapter will make the difference between one class and another. Attributes can be associated with classes, and what could not be done for the individual attribute components, can be done now: specifying attribute values. In case of attributes that are expressed as states, a class should adopt one of the possible states of that attribute. Consider a class that is given the *edibility* attribute. If the class is meant to describe an apple, it should adopt the *edible* state. For the *table* class, the *inedible* state is more appropriate. Attributes expressed as a numerical value will request the assignment of a number. When a value has a dimension, a selection of the unit (and prefix) is required as well; otherwise it is still not clear what the value exactly represents. A *temperature* attribute that has been given the value 20, for example, will be interpreted differently for *degrees Celsius*, *Fahrenheit*, and *Kelvin*.

When attributes have numerical values, and especially when they are dimensionless, a problem could occur if no range of values has been defined, as it is then unknown whether a value is acceptable. Consider a *couch* class with a dimensionless *level of comfort* attribute. A value of 80 will then be meaningless, because it is still unknown whether that couch is comfortable or not. To solve this problem, attributes that are associated with classes can get optional minimum and maximum values. For some attributes, like some physics coefficients, this will not be necessary, while for other attributes, like a *level of comfort* that ranges from 0 to 100, this is quite useful. Of course, when minimum or maximum values are not required for attributes, those values will be negative and positive infinity, respectively. Figure 13 shows a class with an attribute with a value within a range, and an attribute that has adopted one of the two available states.

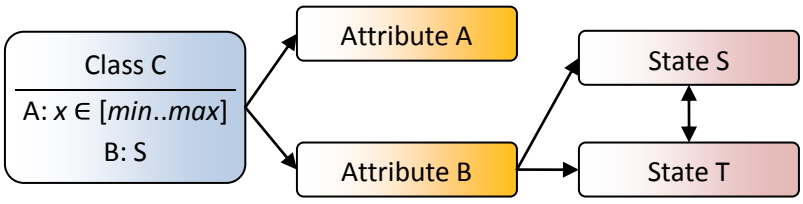

Figure 13. A class with a numerical attribute and an attribute with a state

To define semantics for classes, one or more services can be defined for them. As this is an extensive topic, a whole chapter has been reserved for it (Chapter 5), so more information can be found there.

There is more to classes than attributes and services only. In many cases, an entity is not just one entity, but a collection of entities that together form that entity. This aggregation relationship was already mentioned in Section 2.2. Example assemblies are a table, which has a table top and several legs, and a flashlight, which has, among others, a casing, a light bulb, and several batteries. In this research, classes can have dual part-whole relationships with other classes. From one perspective, a class can be a whole that is assembled of multiple parts. From the other perspective, a class can function as a part that is assembled in one or more wholes. The previous examples were treated from the first perspective; an

example from the second perspective is a battery, with a flashlight and an mp3 player as possible assemblies. When considering classes, a part can be related to multiple wholes, while this is not possible for instances. After all, a class is just a generic description of an entity, while an instance, an in-game 'physical' representation of a class, can be in only one assembly (and one place) at a time.

When discussing wholes and parts from the whole perspective, an extra detail can be added: quantities. Although the *flashlight* class might have the *battery* class as a part, it is relevant to know how many batteries it actually needs. This is relevant for two reasons. First, this can serve as a requirement of a service: only when the *flashlight* is assembled of all its required *batteries*, will it offer its services. Secondly, this description helps game artists in the customization phase to construct their 3D models in the correct way. When defining parts for classes, however, one should be aware of the amount of detail, as it is nearly impossible and impractical to define all nuts and bolts. Still, even with fewer details, wholes and parts can be useful for game objects and services.

Having individual parts gives another advantage. When a material is applied to a class without parts, the entire class will inherit the attributes of that material, as if the entity that is represented by the class is completely constructed of the same material. This makes it impossible to, for example, design a wooden bench with steel, fire-proof, legs, as applying the *wood* and *steel* material to the *bench* class will raise a conflict with the *fire-proof* attribute, assuming that both materials have contradicting values for this attribute. Applying materials to specific parts resolves this problem, because each part will have its own attributes instead.

The discussion about attributes and parts might have raised the question when values and quantities have to be specified. After all, one table might have four legs, while another has three legs. When defining generic classes in the specification phase, one should keep in mind that they are only generic descriptions, in contrast to game-specific classes and instances in the other design phases. Therefore, only values and quantities that are meant for all instances of a class should be defined in the specification phase. Otherwise, one could define temporary default values, which are placeholders before game-specific classes are designed based on a class in the customization phase. Those game classes could then be assigned more customized values/quantities, if this is wanted by the game or level designer. An advantage of having default values is that common game objects are much easier and faster to design.

Finally, units can be specified for classes. At first sight, this seems strange, which is understandable when only considering classes that describe physical objects. One should not forget that entities can be substances as well. If this is the case, it is helpful to indicate in which units those substance classes can be expressed. For this, the *volume* unit category can be used, which contains units like *liters*. Here again, a generic class is not the correct place to define a substance quantity (like how much liters water). Only when creating instances of such classes (when placing 1 liter water in the game world, for example), this is required. More about this can be found in Section 4.6.

4.5 Game-Specific Classes

Game-specific classes are components that are not generic, but specifically customized for each game development project. Because they are used in only one game (series), a 3D artist is able to create a model for them. When an actual game is created, this 3D model will represent an instance of that class in the game world. Besides a model, a game-specific class has its own specific behavior. As most game-specific classes are based on real world objects, they will have some common behavior as well. Because this common behavior has already been defined for generic classes, game-specific classes can be based on generic classes. This way, all common materials, attributes, and services are inherited.

To allow even more object customization (although this will be minimal when the generic classes have been properly designed), new materials, attributes, and services can be specified as well, in the same way as was done with generic classes. As stated in the previous section, not all attributes can be given correct values. A sword in one game will have a different mass from a sword in another game. Game-specific classes are the components that require specific values for the available attributes.

Instead of basing a game-specific class on just one generic class, it is also possible to base it on multiple classes, resulting in more specialized behavior, and providing the possibility to, for example, design a *microwave* that also emits music, by combining the *microwave* and *radio* classes. This could be useful in a game with many fictional and/or magical objects. In Figure 14, a game class can be seen that is based on two generic classes, inheriting their attributes and values. In such a situation, it is possible that a game class inherits attributes with conflicting values (like *edible* versus *inedible*). It is then the task of the game designer to solve this problem, by defining which value should have the upper hand.

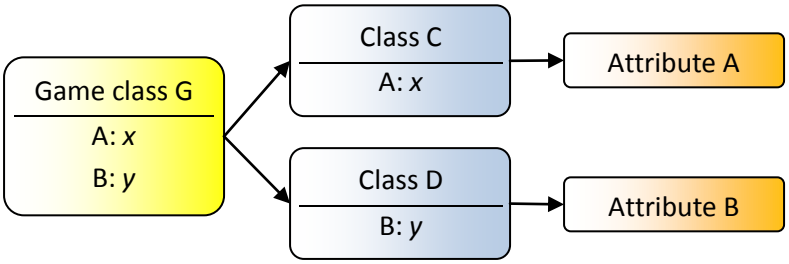

Figure 14. A game-specific class that is based on two generic classes

In the specification phase, generic classes representing an assembly can have other generic classes as parts. In the customization phase, this is extended to game-specific classes: if a game class represents an assembly, it can have other game classes as parts. If a game class is based on a generic class which has a particular generic class as a part, the game class should have a game class as a part which is based on that particular generic class. Consider the generic *table* class with the generic *table top* and *leg* classes as parts. If a *modern table* game class is designed, it can be based on the generic *table* class. This means that there should also be game classes that are based on the generic *table top* and *leg* classes. The main reason for this requirement is because of the use of 3D models, which are assigned to game classes only. When having parts, the *table* does not have just one model, but is composed from the models of its parts. In this case, the models of the parts have to know where they are positioned in the whole model, which can be done with relative positions. A side effect of this approach is that models

should be created for each individual part, which is not always desired for game development teams that want to keep the design phase simple by creating just one model for an entity. Therefore, this research still allows this possibility by making the definition of parts and wholes optional in the customization (and instantiation) phase, even when parts have been defined for a generic class on which a game class is based.

Parts and wholes are also useful for scene composition (see Section 2.5). For example, instead of defining an offer area at the top of a *table*, it is now possible to define that area on the *table top* itself. This way, the *table top* might be used as a part of other game classes as well, without redefining the offer area for each game class. This, however, is not the topic of the current research.

One more model related topic is the use of animated models. When an entity performs a particular action in a game, it is often desirable that this is accompanied by an animation of the entity that visually shows the action. To achieve this, three things have to be done. The first is obvious: the model should have the animation and know how it can be played (like with key framing or inverse kinematics). Of course, to allow the visualization of multiple actions, multiple animations in one model are required. Secondly, the accompanying game class should know these animations, by, for example, deriving them from the model. After that, it will be possible to link an action of the game class to one of its animations. Although unrelated to models, something similar can be done with sounds that should be played when a particular action is performed by an entity.

With the exception of models, animations, and sounds, game-specific classes are still pretty similar to generic classes. There is, however, one more difference, as game classes are also given an inventory. As has become clear in Section 3.2, and will be elaborated in the next chapter, an entity can have the service to supply an entity from its inventory (consider a vending machine with cans of soda). However, in order to do so, the entity should have an inventory in the first place. Initially, the inventory of a game class is empty, but if desired by the game designer, it can be filled with one or more game classes in the customization phase, accompanied by a quantity. For example, a designer might add 30 cans of *orange soda* and 40 cans of *cola* to a *vending machine*. Just as parts in an assembly, an inventory should also be aware of its position relative to the model of its game class. This way, items in a chest, for example, can be placed inside (the model of) that chest, instead of on top of it. Note that this research does not cover the use of multiple inventories for one game class (like a jack with multiple pockets), although this would be a small addition that should not give much design problems.

4.6 Instances

When a game is in development, it is not the classes that are placed inside the game world. Instead, *instances* are introduced, which are single occurrences of a game-specific class in the game world. This can be seen in Figure 15. From one game class, an ‘unlimited’ (theoretically) amount of instances can be created, and positioned in the world by the level designer. As each instance represents the same game class, they will all behave the same. Sometimes, this is unwanted. Consider a level designer who wants to fill a basket with *apples*, of which one is *rotten*. Instead of creating instances from the *fresh apple* and *rotten apple* classes, it is easier to create instances from one *apple* class, and switch the

fresh/rotten attribute of one instance to the *rotten* state. To support this, instances can be customized as well.

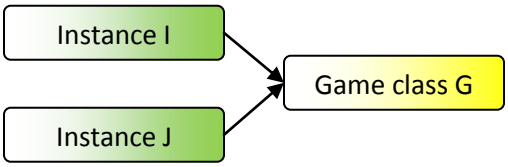

Figure 15. Two instances of a game class

Besides customizable attributes, there are more customizable aspects for instances. An obvious one is their position in the game world. Related to this is their rotation along the three axes, just like the scale factor and transparency value of the model. In the instantiation phase, the parts of an instance are instances as well. So, when an instance is made from a game class with several parts, instances of those parts should be made as well. As far as wholes are concerned, an instance can only be part of one whole, as it can only be at one position at a time. The same applies to the inventory of an instance. Customization of services per instance is possible as well. Although this will be discussed in more detail in the next chapter, a short example can be given. Consider the service of a key, which is unlocking a door. However, as not all keys unlock all doors, there should be a possibility to link the *unlock* action to a specific *door* instance.

An instance represents a single occurrence of a game class in a virtual world. Although this definition is fitting for physical objects, it does not for substances. To solve this problem, instances of the substance type should be given a quantity, together with a unit to express that quantity. This way, substances in particular quantities can be placed all around the virtual environment. However, with substances, the position, rotation, and scale are inapplicable, and it might be easier to make use of advanced fluid simulations. However, it goes beyond the scope of this research to take care of these simulations.

With the previous discussion about instances, all the components have been treated. For classes and instances, an explanation of services was omitted, because the whole next chapter is dedicated to them.

5. A Detailed Description of Services

Chapter 3 introduced the notion of services. As services are easier to explain and understand with components, those components were discussed in the previous chapter. Now, it is time to delve deeper into a service by describing all its elements. At the end of this chapter, it should be clear how generic classes, game-specific classes, and instances can be enriched with services.

A service has been defined as the capacity of an entity to perform an action, possibly subject to some requirements. Services can be declared for generic classes, and customized for game classes and instances. In addition, new services can also be declared for game classes. This chapter will give much more insight into the design of services, by first discussing their optional requirements in Section 5.1, including preconditions, passive actions, demands, and part requirements. As services will be meaningless without actions, which can be found in Section 5.2, along with some extra properties that should be taken into account. This is followed by several final remarks in Section 5.3, such as a sequence of interaction steps. Figure 16 gives an overview of all the requirements and actions of a service. Consulting this figure throughout the chapter might be useful for a better understanding of a service. The general idea, however, is the following: once all the requirements of a service have been satisfied, the service is able to perform its actions.

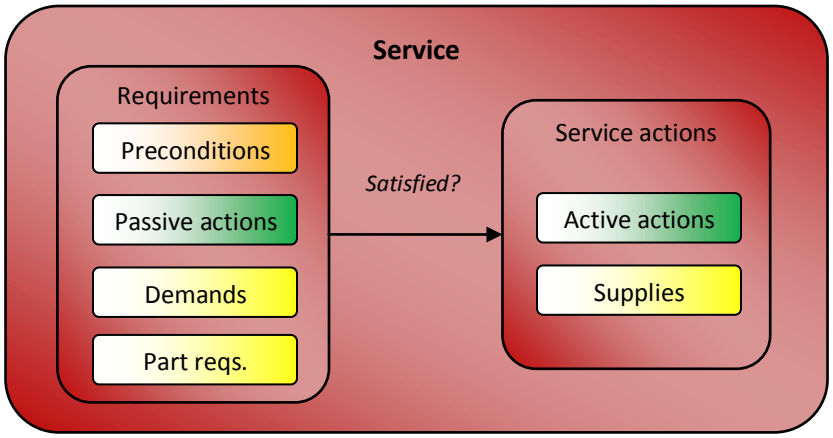

Figure 16. A service with requirements and actions

5.1 Requirements

Many entities have at least one service. Some will offer that service 'for free', while others will have one or more *requirements* in order to trigger the action of that service. When thinking about requirements

and expressing them with components, they can be related to attributes, generic classes, game-specific classes, instances, and actions. In this research, requirements in the form of attributes are called preconditions, requirements in the form of classes or instances are called demands, and requirements in the form of actions are called passive actions. For parts, a special type of requirement has been designed. Each of these requirements will be discussed individually in this section. Note that a service is not limited to requirements of one particular type, as one can also think of a service with preconditions and demands, or any other combination. An example is a vending machine, that requires to be powered on (a precondition), and demands a coin. Also note that demands are considered as special passive actions, which will be explained further in their own subsection.

5.1.1 Preconditions

When an instance is placed inside a game, its attributes will be set to the default values that were given to it in one of the three design phases. When the game is running, these attribute values can change over time, thanks to actions performed by (other) entities. This can either be a numerical value change, or a state change. (For example, a *heater* might increase the *temperature* attribute of instances around it.) Nonetheless, this is where *preconditions* make their introduction, as they are related to these attributes. For each class, a service can be defined with one or more preconditions, which are related to the attributes of that class. (This means that each attribute of a class can be a precondition for all its services.) A precondition requires an attribute to have a specific value (or reach that value during runtime), or quite the contrary, to not have a specific value. If that happens, the precondition is satisfied. Now, if other requirements of the service are satisfied as well, the action(s) of the service will be performed.

When an attribute is expressed with states, the precondition requires the attribute to (not) have a particular state. An attribute can also be expressed with numerical values with optional units. If this is the case, a precondition requires the attribute to (not) have a specific numerical value, which is related to inequality statements. The value could be equal to (==), not equal to (!=), greater than (>), greater than or equal to (>=), less than (<), or less than or equal to (<=) a specific value. These equivalence relations could be extended with AND (&) and OR (|) operators for more complexity. This way, it will be possible to require an attribute value that is greater than 5 and less than 10, for example. Figure 17 shows an instance that is associated with attribute A, having numerical value *x*. For the instance, a service S has been defined, which has a precondition as a requirement, stating that the attribute value should be higher than a specific value *y*, in order for S to be satisfied.

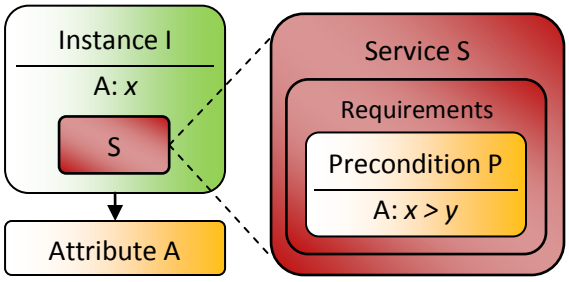

Figure 17. An instance having a service with a precondition

5.1.2 Passive Actions

In Sections 3.2 and 4.3, an action (component) was described as a process that is performed by an entity. This definition described an action from the perspective of the actor, which will be called an *active action*, and will be further discussed in the next section. Possibly, this action is performed on a target. From the perspective of that target, the action is passive, hence the notion of *passive action*. Consider the subtle difference between a person and an apple: the person is able to eat (active), while the apple can be eaten (passive). We would like to let an in-game person 'know' that it can eat the apple. Above all, the apple has to know that it should satisfy the hunger of the one that eats it. Several things can be done. For a *person* (class), one could directly define which entities it can or cannot eat. This, however, is a lot of work and will result in many relations between the *person* and all *edible entities*. If there is another entity that can eat as well, like an *animal*, all these relations should be defined again. To solve this problem, it is also possible to define for the *eat* action component which entities can be eaten, and associate that action with all entities that are able to eat. This is an improvement, but still results in many relations between entities and that action. Furthermore, it raises the question where the effect of eating an apple should be defined, as each edible entity could have another effect. Defining this for each entity at the *eat* action component is an undesirable solution. Furthermore, the apple itself would not know that it can be eaten, as only the *eat* action knows that!

To overcome these problems, we define two different services. One service is for the *person* class, having the *eat* action component as an active action, indicating that it is able to eat. The other service is for the *apple* class, having the same *eat* action component as a passive action (requirement), indicating that it can be eaten, and an active action that is performed when that requirement has been satisfied (satisfying the hunger of the person when he eats it). For every entity that is able to eat, a service similar to the one of the *person* can be designed, as a service similar to the one of the *apple* can be designed for other edible entities. Figure 18 shows an abstract view of the relation between active actions and passive actions. For simplicity, effects are omitted here. Class C can perform an active action, based on action X. Class D knows that this action can be performed on it, because it has a passive action that is also based on the same action X.

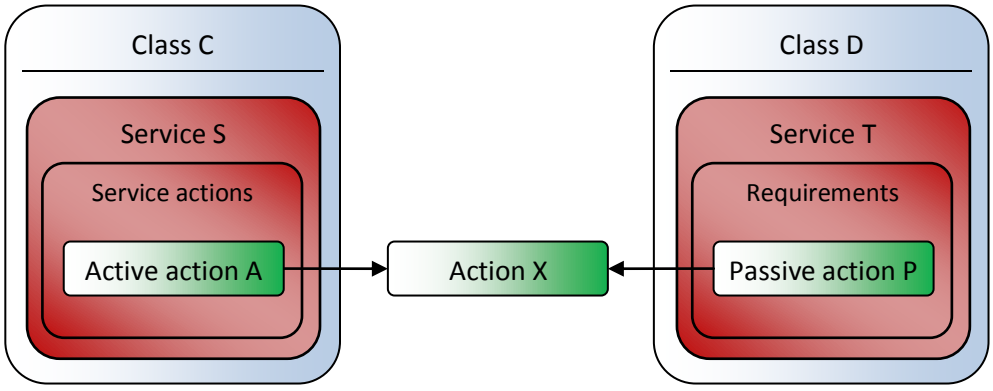

Figure 18. A class with an active action, and a class with a passive action

In case of the *apple*, the *eat* action might satisfy the hunger of the one that eats it. In case of a *coat*, it might provide warmth to its wearer. Because for the action component has been defined whether it is discrete or continuous (introduced in Section 4.3, further explained in Section 5.2), the passive action will be discrete or continuous as well, meaning that the passive action is only satisfied at a particular

moment, or satisfied for a longer period of time. This affects the actions of a service, as they are triggered only once, or for a longer period. Furthermore, it might be useful to define for a passive action what should happen when all service requirements (including the passive action) have been satisfied and the corresponding actions have been performed. The passive action could remain satisfied, or it could be reset, indicating that it should be satisfied again in order for the service to be triggered again.

In the process of an action, it is also possible that the entity with the service with the passive action is exhausted, which was defined at the action component. This prevents the apple from being eaten a second time. However, care should be taken here, as the apple should first perform its own actions, before being exhausted.

Passive actions can be combined with preconditions in order to indicate when they can be performed on the entity (class or instance) with that passive action. This is, for example, useful when designing a door that can be opened, but only when it is in the *unlocked* state. As discussed in the previous section, these preconditions are based on the attributes of that entity. It should, however, also be possible to related passive actions with preconditions that are based on the attributes of the actor of that passive action. There is one problem, though, as it is unknown until run-time who or what that actor is. Therefore, these preconditions cannot be retrieved automatically from the actor's attributes, but should be defined by the game designer. The *pickup* action is a good example. Each *physical object* can be picked up (defined as a passive action), but it depends on the *strength* of the actor whether this succeeds or not. Therefore, a precondition can be defined that requires a particular value for the *strength* attribute. Dependent on the *mass* of the object, the designer can decide to make this value high or low.

5.1.3 Demands

The third type of requirement is a *demand*, a term borrowed from the economic supply and demand model that was discussed in Section 3.2. Demanding and supplying 'something' are treated as special actions in the definition of a service, as although they do not affect any attribute, they are involved in the interaction between two entities. With demands and supplies, the inventories of both entities play a key role instead, as 'something' that is demanded by entity A can be supplied by entity B, by removing it from its inventory, and giving it to A, after which A adds it to its own inventory. It should be clear that this 'something' is another entity, represented by an instance when this exchange is performed in a game. From the perspective of A, this entity will be called the *demand*, while it will be called the *supply* from the perspective of B. Demands are considered requirements of a service, as when they are satisfied (when they are given by another entity), the entity with that service can perform the corresponding action of a service. Although entities are represented as instances in a game, demands do not have to be a specific instance when they are being defined in the design phases. Consider a coffee machine that demands coffee beans. If the machine demands one specific *coffee bean* instance, it is impossible to satisfy the requirement with other coffee beans. Therefore, a demand can also be defined as an instance of a particular game-specific class. Now, each instance of a *coffee bean* game class is accepted. A similar reason can be given to explain why demands can also be represented by generic classes.

In the specification phase of the object design process, demands can only be represented by generic classes, as no game-specific classes and instances are known at that point. In the customization

phase, game classes inherit the services of the generic classes they are based on, after which the demands could be further customized to be a particular game class, instead of a generic class. Of course, that game class should be based on that generic class, as it would contradict the generic service otherwise. Finally, in the instantiation phase, the demand could be further customized to be a particular instance of the game class.

There are three possible ways for an instance to get a demand in its inventory. First, it is possible that the demand is supplied by another entity during runtime. Second, the game or level designer could already have added the demand in the inventory during the customization or instantiation phase, as was discussed in Sections 4.5 and 4.6. Finally, when the instance represents an intelligent creature (controlled by artificial intelligence in a game), or when the instance is controlled by the player, the demand might be picked up and added to the inventory. See Chapter 7 for more information about this special pick up action.

When demands are defined for services of classes, quantities are essential to indicate how many of them are required. When the demand is a physical object, the quantity will be an integer, while a floating-point value with a *volume* unit is required for a substance. Only when an entity has the correct quantity (or: amount) of a demand in its inventory, the demand will be satisfied. An example is a *vending machine* that demands two *coins* to supply its *soda*. (Note that the actual value of these coins is not taken into consideration here.) An abstract version can be seen in Figure 19, where a service has a demand that is a game-specific class in a particular quantity *x*.

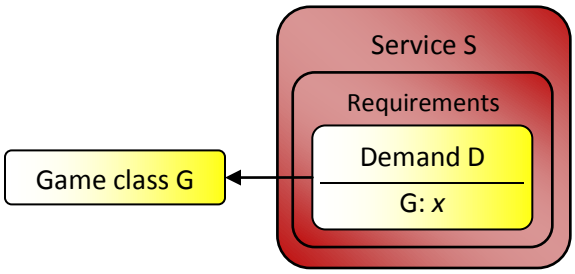

Figure 19. A service with a game class as a demand

In real situations, a quantity does not have to be an exact value. Although this might be wanted in some situations, it might form restrictions in other situations. To illustrate this, demands can be compared to the ingredient list of a recipe. If it is possible to make an apple pie with, among others, 4 apples and 300 grams of flour, it is also possible with multiples of these quantities. In this situation, proportions are required instead, and the demands are satisfied when an entity has those demands with the right proportions in its inventory.

Earlier this section, preconditions were defined as requirements of a service, but in addition, demands can also have preconditions of their own. This makes it possible to specify a demand in more detail. Preconditions are now not based on the attributes of the entity that provides the service, but on the attributes of the demand (class or instance). For instance, *coffee beans* as demands of a *coffee machine* can have a precondition to be *fresh*, instead of *rotten*.

So far, almost all elements of demands have been discussed. There is, however, one final aspect that should be kept in mind when defining them. One should indicate what should happen with the received demands when the action of the service is actually performed. In the *coffee machine* example, there should be no *coffee beans* left when the *coffee* is made. However, the *coins* that were inserted in a *vending machine* should remain in its inventory when it supplies a *soda*. As can be noticed, there is a difference in the usage of a demand. Therefore, when a demand is defined, one should indicate whether the demand is *used* only, or *exhausted*. In the former case, nothing happens with the demand, while in the latter case, the correct quantity of the demand is removed from the inventory (and even from the game, as it should not exist anymore). For example, when 0.2 liters *water* are demanded by a *coffee machine*, but 0.5 liters are given, only 0.2 liters will be exhausted, and 0.3 liters are left over. For demands that are used only should also be defined whether they should be received each time, or whether they should just be present in the inventory. For example, a *tea bag* in a *cup of tea* is only used when *water* is added to make *tea*. When refilled afterwards, the same *tea bag* can be used again. For a *vending machine*, the demanded *coins* should be given always. Otherwise, each subsequent *soda* will be free, as the inserted *coins* for the first *soda* are still present in the machine's inventory.

5.1.4 Part Requirements

The final type of requirement is a part requirement. Although parts and wholes were described in the previous chapter, they did not have any specific use, besides the construction of 3D models and the split-up of attributes of an entity. However, they are also very suitable for the requirements of a service. For this, there are even two possibilities.

The first possibility is requiring that an entity that represents an assembly consists of all its parts, before being able to perform the actions of the corresponding service. An example is a car that should have an engine before it is able to drive. Although a designer can define in the specification and customization phase that a car should have one engine, this does not automatically mean that a car instance in the game world should have one. A level designer might decide to remove the engine from the car and place it somewhere else in the world, so a player should search for it first, before being able to attach it to the car and drive away. For each part a class has, a part requirement can be defined for the services of that class, just like each attribute can be a precondition.

The second possibility is requiring for a whole that a particular service of one of its parts should be satisfied, before the requirements of that whole are satisfied. Electrical devices with buttons might have such requirements. Consider a television with an on/off button as a part. One service of the TV might require the button to be pressed (a service of the button), before the TV performs its own action of switching on (or off).

5.2 Actions

The action components in the action library represent all the different actions that can be performed, and gives a general description of the effects of those actions. That does not immediately mean that each entity can perform them, as classes have to be related to them first. The action part of a service is the place to establish these relations. Once the requirements of an entity's service in a game have been satisfied, the entity will perform all the actions that are associated with this service. This section will

discuss these so-called *active actions* (to prevent confusion with the action components), and a special active action that involves the supplies that were mentioned in Section 3.2. Furthermore, spatial and temporal properties are discussed, which indicate who or what is affected by a service, and what the duration of an active action is, respectively.

5.2.1 Active Actions

As active actions are not that different from action components (which were already discussed in the previous chapter), there is not much left to tell about active actions. Active actions represent all the actions that an entity can perform, possibly subject to some requirements, defined in the same service as where the active action is situated. As active actions are based on action components, which was shown in Figure 18, they will already know what their effects are when they are performed. The effect could change the state of an attribute, or increase or decrease an attribute value. For the latter, the specific value is still unknown. Therefore, a game designer should specify the exact change in value here, and if needed, the corresponding unit as well. If this value change is dependent on space and time (because the action can be continuous), spatial and temporal properties can be used, which will be described in Subsections 5.2.3 and 5.2.4. Of course, when the action is performed on an entity that does not have the affected attribute, the effect will do nothing. Performing the *open* action on an *apple*, for example, has no use, as it does not have an attribute with the *opened* and *closed* states.

In some situations, the change in value of the effect cannot be decided beforehand, as it might be dependent on one of the attribute values of the actor. Consider a *microwave* with a button to adjust the amount of *power*. The higher that attribute value, the higher the *temperature* of its *heat* action should be. This, however, requires the attribute to have a minimum and maximum value, and the effect value to have an initial value that can be multiplied by the *amount of power*. For example, suppose that the *amount of power* is 700 W, with a minimum of 0 W and a maximum of 900 W, and the initial effect value is 100°C. The resulting effect value would then be 100°C * (700 W / 900 W) = 77.8°C.

Games and probabilities seem to go hand in hand, ranging from the classic board games, games of dice or other games of chance, to next-gen computer games. A game like *Neverwinter Nights* (BioWare, 2002) is heavily based on chance. Whenever a spell is cast or an attack is executed, calculations of probability decide how effective the spell or attack is. To support this functionality, chances (ranging from 0 to 1) have been defined for both service actions, and the individual effects of active actions. For effects, it indicates whether they will be triggered. This way, it is possible to, for example, design a *health potion* that always restores *health* (chance 1), but could have a *stunning* side-effect (with chance 0.1). In this case, the chances were independent from each other, but it is also possible to link them, making it possible to design a *potion* with a chance of 0.9 that it restores *health*, and a chance of 0.1 to get *stunned*. It should be obvious that the sum of all dependent chances should be equal to 1. For service actions, the chance value indicates the chance that the action will be performed. Here, chances can be coupled to attributes, allowing, for example, the chance to succeed in *lock picking* to be dependent on the actor's *lock picking skill* (attribute), just like in ***Oblivion***. Suppose the *lock picking skill* of the actor is 40, having a minimum of 0 and a maximum of 50, and the chance of success in *lock picking* is 0.1 times the skill level, then the actual chance to succeed is: 0.1 *(40 / 50) = 0.8.

When an active action is being customized for a game-specific class or instance, it is possible to link it to an animation. As generic classes are not aware of models and animations, this cannot be done in the specification phase. Of course, the animation has to be defined for the game class or instance first. When an instance then performs the active action in a game, its model will play this animation to visually represent the action. Dependent on whether the action is discrete or continuous, the animation will play once or loop. This action time is also needed to indicate how often or how long the action should be performed when it is triggered. For discrete actions, an active action has a frequency, so it can be performed once, or two or more times after each other, if desired. For continuous actions, a duration should be specified, which will be further discussed in Subsection 5.2.4.

5.2.2 Supplies

Supplying an entity to another entity is considered a special action of a service. Supplies are the opposites of demands, and do also involve the inventory of an entity, instead of its attributes. An introduction of demands and supplies was already given in the previous section. The three-phased game object design process has three moments where supplies can be defined. In the specification phase, a service can be defined for a generic class, after which another class can serve as a supply for that class. For example, the *vending machine* class can have the *soda* class as a supply. In the customization phase, it is then possible to design game classes, and customize the supply a bit further by defining which specific game class represents the supply. Just like with demands, only game classes that are based on the same generic class as the supply will be valid candidates. This is shown in Figure 20. In the specification phase, generic class C has been defined with a supply that is based on class D. If, in the next phase, game class G is based on class C, it should supply a game class that is also based on class D. In this case, game class H is a valid supply.

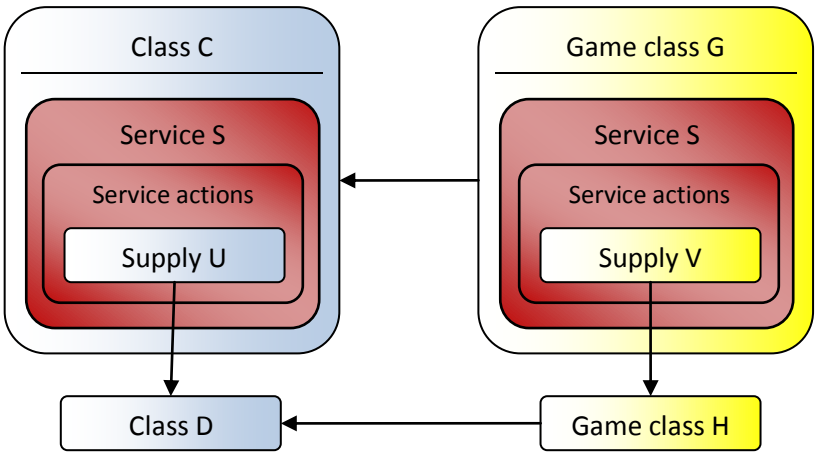

Figure 20. A game-specific class with a supply, based on a generic class

For a particular *vending machine* game class, the service of supplying a soda can be more customized by defining which *soda* game classes can be supplied. Finally, in the instantiation phase, it is then possible to create instances of the game classes and define for a service which specific instance is supplied. However, in this example, this is undesired, as it does not matter which specific instance of a *soda* is supplied to the user of the *machine*. Instead, it is sufficient to supply a random instance from the inventory, as long as it is based on the correct game class.

The last example showed that supplies are related to the inventory of the entity that gives the supplies. Before an entity supplies a particular instance to a target, it has to be inside the inventory of that instance. Once it has been given, the supply will be put in the target's inventory. How the actual transfer of the supply is visually represented in a game is beyond the scope of this research. On clearer inspection of real-life entities and objects in current games, one might notice that the inventory is not always involved, as supplies can be produced as well. Consider a coffee machine, which exhausts the coffee beans and water, and gives coffee in return. In contrast with the vending machine, the coffee was not stored inside the inventory of the coffee machine, but created when the service was activated. In real-time strategy games, military units are also often magically created when the correct resources are given to a factory. To support this, supplies can either be taken from the inventory, or newly created.

Just like active actions, supplies are also equipped with chances, to indicate the probability that a supply is given. And just like demands, supplies are accompanied with quantities. Although most instances only supply one instance from their inventory, there may be some that supply more. Therefore, quantities can be specified in the first two phases of the design process. In the instantiation phase, this cannot be done, as it is impossible to supply a specific instance more than once. Instead, a designer should choose which specific instances from the inventory should be supplied. In some cases, quantities are undesired at all, as it is also possible that all inventory items of a specific type should be supplied, independent of the quantity. For example, a fruit machine might supply all its coins when the jackpot is won. To allow this customization, the design provides the selection of either a quantity, or all items of a specific type.

5.2.3 Spatial Properties

An action is performed by an entity, the actor, and has one or more effects when it has been performed. A target is required that is affected by those effects. Thus far, it was assumed that an action can only be performed on one specific target, or a supply is only given to one specific target. This, however, is not a prerequisite, as there can be multiple targets as well. Therefore, *spatial properties* should be defined, which, among others, make it possible to select the possible targets:

1. First of all, the target of an action can be the entity itself: the *actor*. An example is a person that heals himself/herself.
2. It is more common that an action is performed on a chosen target (e.g. a person that punches a punching ball), which means that the target can also be *any entity*, different from the actor.
3. Instead of any entity, it is also possible that an action can only be successfully performed on a *specific entity*. A key, for example, will usually only unlock one specific door.
4. A fourth possibility is that the target is the entity that triggers a service, called the *trigger*. Consider a vending machine, which demands an entity (the user) to insert some coins, after which it supplies a soda in return. In this case, the user is the trigger.
5. The *entities inside the inventory* of the actor can also be the targets. For instance, a refrigerator has the service of cooling down the entities that are stored inside it.
6. Finally, an action can also affect all *entities within a certain range* of the actor. In that case, the numerical value of that range and the length unit for the value are two extra elements that can be defined.

There is something that should be taken into consideration when the last option is used for effects with numerical values (not for effects with states). Consider a *heater* that spreads its warmth to entities around it. For simplicity, one might define that each entity within a radius of, for example, 5 meters is heated up for 20 degrees Celsius. This, however, is not what happens in the real world, as heat with a particular initial temperature is emitted from the source and spread around it with a more decreasing value the farther away it is from that source. A specific radius it not defined here, and the decrease in value is modeled by a mathematical function, which should stop when the value has reached 0. As this research is about games, it is questionable whether this deep level of detail is desired, because it will probably not affect the gameplay. Still, it is a wise idea to provide this option, in case a realistic simulation is desired (or required) in, for instance, a serious game where fire plays a central role. With a mathematical function over space, the initial value of an effect is changed over space. One could think of several functions. The easiest function is a constant one, which is useful when the value should stay constant over space. A linear approach would, on the other hand, be more realistic. Furthermore, one could think of even more functions, such as logarithmic, exponential, or power functions. Figure 21 shows these functions. Note that the values on the axes were left out on purpose, because these functions are just some examples and only their shapes are relevant here. In the figure, the effect values decrease over space (with the exception of the constant function), but by adjusting some function parameters, a designer would also be able to create increasing functions, or modify the current curves.

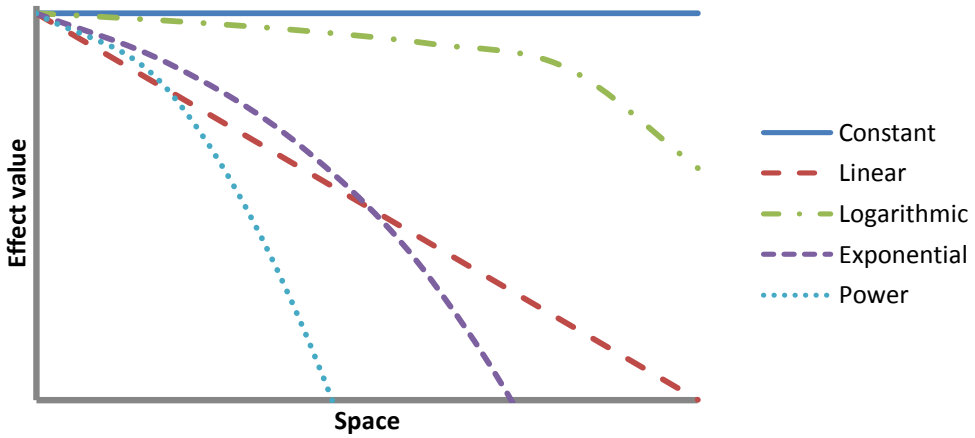

Figure 21. Several functions to change the value of an effect over space

5.2.4 Temporal Properties

Besides spatial properties, there are also *temporal properties* that should be taken into account for each action of a service. For each action component, it is possible to define whether the action is discrete or continuous. In a service, this behavior is extended to active actions. When an action is discrete, only the frequency is relevant. For a continuous action, there are four possibilities:

1. The first possibility is that an active action is performed *as long as the requirements are being satisfied*. An oil lamp, for example, will burn as long it has any oil.
2. A continuous action can also last for a *fixed amount of time*. In this case, the value of this duration should be specified, together with a unit of time.
3. The third possibility is that the action lasts for an *infinite amount of time*, that is, infinite as long as the game is running (or better: as long as the game world exists, in case of massively

multiplayer online games). For this possibility, it is irrelevant whether any requirement becomes unsatisfied at some moment. An example is the sun that shines for an 'infinite' amount of time.

4. For the final possibility, the actor should have some intelligence (either artificial or the player's), as the entity can then *decide to stop the action* at any time. Walking, for instance, is a continuous action that requires the entity's intervention to stop it.

These temporal properties are also applicable to supplies. Consider, for example, a *tap* that should supply *water* as long as it stays *open*. In either case, there is something else that should be defined for temporal properties: an interval. For the *tap*, one might define that it should supply 0.1 *liters* as long as it stays *open*, but if no interval has been defined, it is still unclear when that *water* should be supplied. Therefore, an interval value (with a unit of time) should be specified as well, to indicate, for example, that those 0.1 *liters* should be supplied each second.

In case of a fixed or continuous duration, something similar to spatial properties can be defined: the change of effect values over time. For this, the same mathematical functions (as in Figure 21) can be used, although the space element should now be replaced by time. The longer an action is active, the higher or lower the effect value could become. Here, the *heater* can be used as an example as well: when the *heater* is turned on, it will *raise* the *temperature* of the area, but because the *heater* was cold before it was *switched on*, it first needs some time to reach its desired *temperature*.

5.3 Final Remarks

In the previous sections, services were discussed in much detail. For both requirements and actions, an extension can be made in the form of a *sequence of interaction steps*, based on the usage steps that were described in Section 2.4. A sequence of interaction steps for the requirements of a service indicate in which order those requirements should be satisfied before the service actions are performed. An example is a *music player*, where the user first has to *press* the *power button*, before it is useful to *press* the *play button*. Actually, this sequence is only relevant for demands and passive actions, as preconditions are just attribute values that need to have a specific value. With a sequence, requirements in the first step should be satisfied first (passive actions should be performed on the entity, or specific demands should be received), before the requirements of the second step are being satisfied. If there are several requirements of which the order is irrelevant, they can be placed in the same step. In Table 1, an example can be found, where the order of requirements A, B, and C are irrelevant. However, these three should all be satisfied before requirement D is satisfied. Only after this, requirements E and F are allowed to be satisfied in any order. If this happens correctly, all requirements have been satisfied, after which the entity of the service can perform its action(s).

Table 1. An example sequence of interaction steps

Step 1	Step 2	Step 3
Requirement A Requirement B Requirement C	Requirement D	Requirement E Requirement F

If, during run-time, requirements A, B, and C have been satisfied, and E is satisfied afterwards, instead of D, the sequence is satisfied in the wrong order and will therefore be reset. In this case, one should be aware of the possibility that A, B, or C could be a demand. Because the demand was already given to the entity, and the demand is therefore stored in the inventory of the entity, it is questionable whether the entity should receive another demand of the required type. Consider a *coffee machine* for which, by mistake, has been defined that *coffee beans* are demanded in the first step, *water* in the second step, and a *press* on a *button* in the third (instead of *beans* and *water* in the first step and the *button press* in the second). If *water* is given to the *machine* first, and *coffee beans* afterwards, it is illogical to reset the sequence, as *pressing* the *button* should still result in *coffee*. Therefore, it should be defined for demands whether they should always be given, whatever happens, or just need to be present in the entity's inventory.

The sequence of interaction steps for service actions is similar to the one of the requirements, as it defines in which order the actions should be performed. All actions could be performed simultaneously in the first step, but more steps are possible as well. In the latter case, actions in a later step will be performed once the actions in the previous step have been performed. An exception to this is an action that lasts for an infinite amount of time, as no actions can be performed after it. Therefore, actions after an infinite action will start simultaneously.

There is one final remark that can be made about services. When observing the requirements and actions of a service, and combining some of their elements, one might spot four common different types of services:

1. An *exchange* is similar to the supply and demand model of Section 3.2. In this situation, the requirements are one or more demands, which are exchanged for one or more supplies.
2. An *act* can be compared to the law of action and reaction. Possibly subject to passive actions, an entity performs at least one active action.
3. A *mixture* is similar to an exchange, but for substances only. Here, one or more demands are mixed together, resulting in a particular supply.
4. A final combination involves a *passive action with one or more preconditions*, indicating that an action can only be performed on an entity when the preconditions of that entity are satisfied.

Of course, a service can also consist of another combination of requirements and actions, or just all of them, like in Figure 16. These common service types are therefore not that important, but they might be useful to easily distinct one service from another during the object design phase. The use cases and scenarios that will be given in Chapter 9 are based on these types.

This chapter gave all the details about services, which concludes the design part of this thesis. The following chapters will give details about the implementation of a prototype that actually puts services to use for game development.

6. Prototype – Library and Level Editor

The previous chapters elaborated on an approach to design semantic objects by presenting and discussing several components, and combining them with the notion of services. To show that the given approach is more than an idea only, a prototype has been created that assists game designers to improve object interaction. The prototype consists of two subsystems. This chapter will discuss the first subsystem, which has been made for game development purposes. The second subsystem, for in-game purposes, will be discussed in the next chapter.

The first part consists of two editors, based on the three-phased methodology that was presented in Section 3.3. First, Section 6.1 explains the use of two editors. A description about the first editor, the Library Editor, is given in Section 6.2. Details about the second editor, the Level Editor, are then presented in Section 6.3. Finally, details regarding several shared implementation elements can be found in Section 6.4.

6.1 The Use of Two Editors

Section 3.3 proposed the idea of creating libraries in the three-phased methodology. For the specification phase, this corresponds to the creation and modification of generic libraries of classes, actions, materials, attributes, unit categories, units, and states. The goal of the customization phase is to design game-specific classes, and store them in a corresponding game-specific class library. This way, they can be used in the instantiation phase. To keep track of the instances that are created in this phase, a game-specific instance library is required. Although Section 3.3 already discussed who should design these libraries, there are no tools to actually do this. To overcome this lack, this research introduces a prototype that includes these tools. Although it is possible to throw all required functionality in one big application, this is not a nice solution. After all, the generic specification phase is unrelated to the game-specific customization phase, which, in turn, is independent from the creation of a level or game world in the instantiation phase. This suggests that three applications are needed, which is correct. However, for prototype purposes, only two editors have been created. The *Library Editor* has been made for the first two phases and the *Level Editor* for the third phase. Although the combination of the first two phases might sound odd at first, they have more similarities on closer inspection.

As the name implies, the Library Editor makes it possible to edit libraries. The idea of this editor is to create new components, and modify or remove existing ones. To do so, the editor provides the user an overview of the available libraries, and the components that populate them. Each component can be

modified, meaning that its semantic information can be changed to the user's heart's content. This information includes the relations between that component and other components (possibly of different types), as was discussed in the previous chapters. To support this functionality, the editor allows the user to establish, or remove, relations between components. Furthermore, it is possible to specify quantities and values that are required for some of these relations. Above all, services can be defined. All this functionality, and more, is present in the Library Editor. Game designers are considered the end-users of the editor, and to make the design of semantic objects for them as easy as possible, usability was one of the aspects that was aimed for. In the editor, this has been achieved by providing a clear overview of all information and the possibility to hide unwanted information, and by providing user-friendly ways to quickly establish and remove relations, and edit other semantic information. As each game designer has a different method and background, flexibility is another aspect that was aimed for. This already became clear when the object-centric and component-centric object systems were compared in Section 2.1 (as each designer prefers a different method), and units were discussed in Section 4.1.1 (as each culture might adopt a different base unit). By giving several options to establish relations and define quantities and values, more flexibility is achieved.

The aforementioned is applicable to the Library Editor for both the specification and the customization phase. There is, however, a noticeable difference, as it is not possible to design game-specific classes in the specification phase. After all, the specification phase is generic and not game-related. In an optimal situation, there exists only one set of generic libraries (created once with a Library Editor without the functionality to design game-specific classes), which can be accessed by each development project in order to create their game class library (with a slightly modified Library Editor that allows the design of game classes). For prototype purposes, however, this is not the case, and the generic and game class libraries are just coupled to the same editor, which is further discussed in the next section.

The instantiation phase, in which instances are created, is significantly different from the previous phases, as it does not allow the establishment of semantic relations anymore. Instead, the goal of this phase is to modify the values and quantities of created instances, and place them in a (3D) virtual game world. As this is very game-dependent, the Library Editor is unsuitable for these purposes. Instead, a regular level editor would be more appropriate, because it is based on the game itself and therefore has a visualization of the game's graphics (such as the terrain and models). As creating the actual game worlds is not the focus of this research, the choice was made to not integrate instances with services into an officially released level editor, in order to prevent issues that might have been arisen. Instead, a custom Level Editor has been created, which has very limited and simple level editing options, but is more focused on the adjustment of the semantic information of instances, as was discussed in Section 4.6. More details about the Level Editor are provided in Section 6.3.

6.2 Library Editor

This section will give all the specific details of the Library Editor.

6.2.1 Designing and Relating Components

When the Library Editor is started, the generic libraries are loaded, one for each type of generic component. When the editor is up and running, it is possible to select a component from one of the libraries to view its semantic information, which can be seen in the screenshot in Figure 22. This assumes that there are already some components in the libraries. General information includes the name and description, while more specific component information includes an overview of the attributes or services of a class, the units of an attribute, the related states of a state, etc. Besides viewing information of available components, new library components can be created as well, and existing ones can be removed.

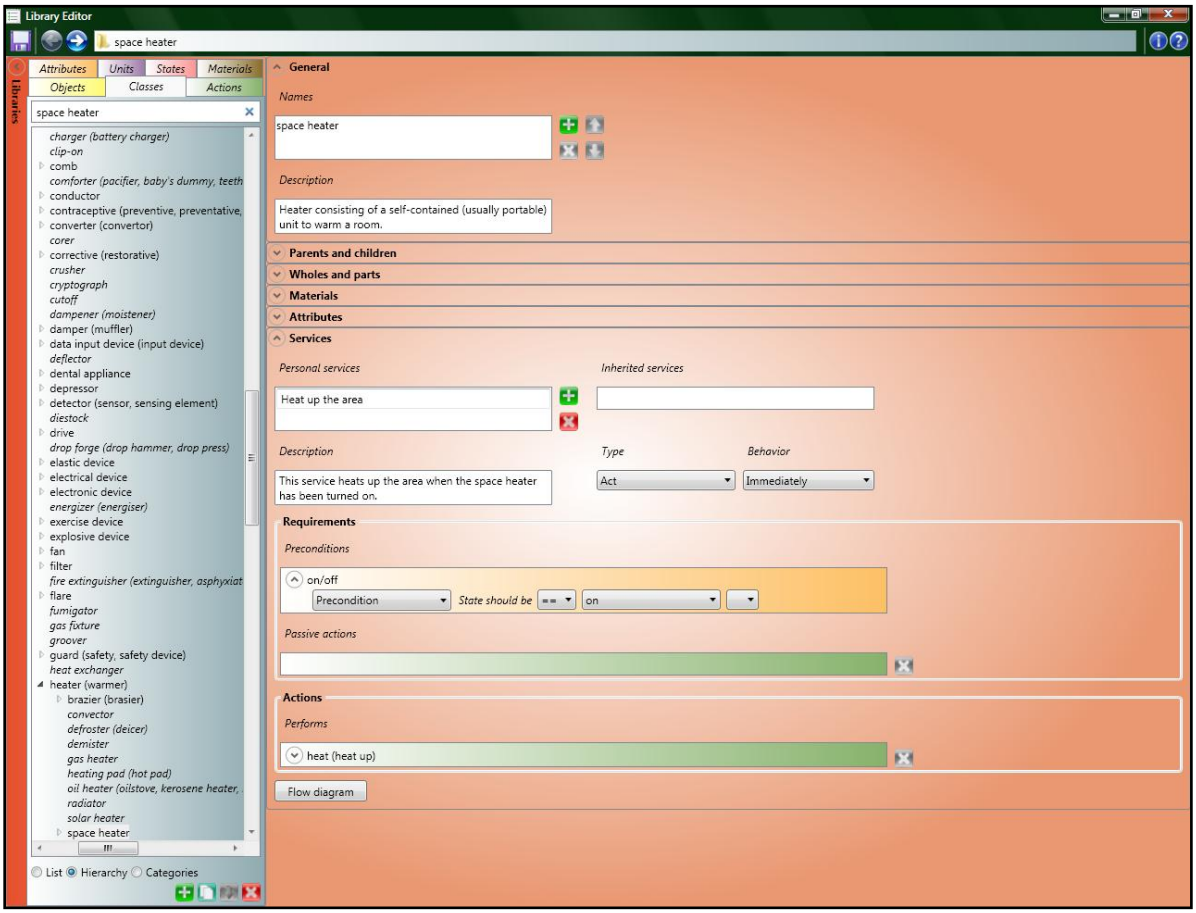

Figure 22. A screenshot of the Library Editor

It is possible to define relations between components by simple and intuitive drag and drop operations. For example, selecting a material from the material library shows a list of its attributes. To assign another attribute to the material, an attribute component from the attribute library can be dragged to (and dropped in) this list. This way, a relation is established, making the material aware of its new attribute.

Besides relations, values and quantities can be defined. The latter is required for wholes and parts, as the whole should know the quantity of a particular part. Figure 23 shows this for a *car*, which is related to several parts, each having a particular quantity. Values are, among others, necessary for

attributes. In the Library Editor, it is possible to relate a unit category to an attribute to indicate in which units its value can be expressed. When that attribute is associated with a class, the designer has the possibility to not just enter the value of that attribute, but also select one of the available units. Figure 24 shows this for the *mass* of a *car*. Furthermore, it is possible to define services for a class. First, one of the service types should be selected (see Section 5.3), after which class and action components can be dragged to the lists with requirements and actions of the service. Then, it is possible to, among others, define quantities of demands and supplies, or temporal and spatial properties of active actions. Figure 25 shows that a *vending machine* can supply one *soda can* from its inventory.

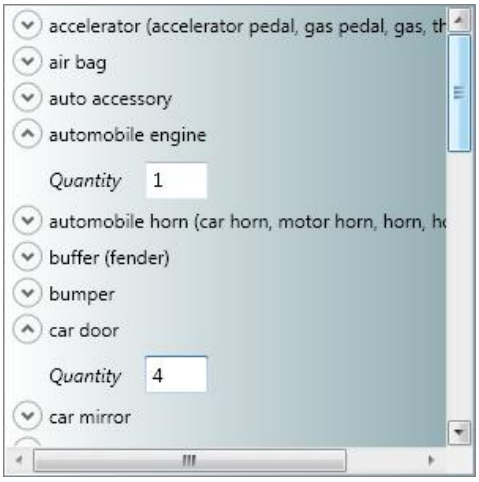

Figure 23. Some parts of a car

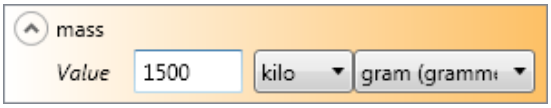

Figure 24. The mass attribute of a car

Figure 25. A supply of a vending machine

The idea of the customization phase is to create a new game-specific class library (or load an existing one), and add new game classes to it. As each game is different, each game development team will create their own game class library. For each game class, the behavior can be defined. This can be done from scratch, by relating game classes with materials and attributes from the generic libraries, or by defining services for them. Better and easier is to drag a generic class component to a game class, in order to let the game class be based on that generic class. This makes the game class inherit all semantic information of the generic class component, assuming that this information was defined during the specification phase. If wanted, a game class can be based on more generic classes, to inherit even more semantic information. Either way, it is still possible to customize all default values and quantities afterwards. (For example, a *Ferrari* game class can be based on the generic *car* class, but instead of the inherited default *mass* of, say, 1000 *kilograms*, a value of 1500 might be desired.) When the game class library is finished (that is, when all desired game classes have been created), it can be saved and used later on in the Level Editor of the instantiation phase.

The next subsection will discuss a special feature of the editor in the specification phase to decrease the time to design generic components.

6.2.2 Generalization and Inheritance

When the generic *apple* class has been properly designed, it offers the service to satisfy one's hunger when it is eaten. Because a pear satisfies one's hunger in real life, the same service will be defined for the *pear* class. Now consider bread, cheese, and tofu, all again having the same generic service.

Defining this service individually for each type of food is a cumbersome task. Much work and time is reduced when the service can be defined once for the *food* class, after which each type of food inherits this service. As the generalization and inheritance ontology has already been discussed in Section 2.2, this notion should be familiar. By applying the generalization ontology to the specification phase, a class becomes the parent of one or more subclasses, the child of another class, or both. From this, a class hierarchy can be deduced, which consists of classes with parent-child relations. Of course, it is not possible for a class to have a child that is already a parent of that class, as this would result in a circular hierarchy. Ordering all classes of the specification phase with parent-child relations results in a huge hierarchy, which can be visually represented by a tree with many branches and levels. Figure 26 shows two abstract relations in a tree. The parent class/node has two children, and both children are aware of their parent. Considering the definition of a class in this research, the one and only root node (the node without a parent) of the complete tree is the *physical entity* class. The *apple* class, for example, could then be reached by walking down the tree from the *physical entity* class to the *physical object* class, and so on, to the *food* class and the *fruit* class, the latter having the *apple* class as one of its children.

Applying inheritance results in children inheriting behavior of their parent. This way, materials, attributes, and services that have been associated with a class are inherited by all its children. Therefore, an attribute like *mass*, for example, does not have to be defined for each single class, but to the *physical object* class only. Specifying classes becomes much easier this way. In Figure 27, an abstract example shows that for parent class C, a service S has been defined. Because of inheritance, C's children also inherit this service (including requirements and actions). The services of a class are not limited to the services of their parent, though, which can be seen for class E, having a personal service T in addition to inherited service S.

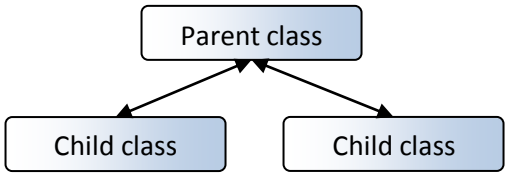

Figure 26. Parent-child relations

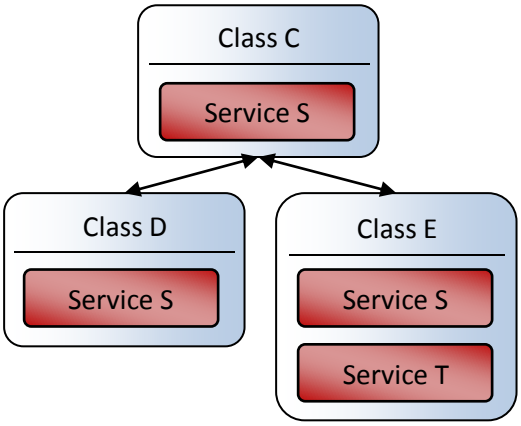

Figure 27. Child classes inherit services of the parent

It should be obvious that the *table* class, for instance, is positioned in a completely different branch of the hierarchy than the *apple* class. However, in which branch should an electrical heater be placed? This entity is both a heater and an electrical device. Because a heater does not have to be an electrical device, making the *heater* class a child of the *electrical device* class is not a solution. Neither is the other way around. To overcome this problem, multiple inheritance can be used. This way, a child can have more than one parent and inherit more attributes and services. Although this is very advantageous, there is also a problem that can arise. There is the possibility that two equal attributes, inherited from two different parents, have conflicting values (a similar situation was encountered in

Sections 4.4 and 4.5, where a class could have multiple parts with the same attributes). Although this is a major ambiguity issue for programming purposes (better known as the *diamond problem*), it is easier to cope with here, because the correct value can still be chosen with user intervention.

The generalization and inheritance ontologies have been implemented in the Library Editor: for each generic class, parent and child classes can be defined. Associating an attribute to a class will make each child inherit this attribute, together with the value that was given to it. Although it is not possible to remove an inherited attribute from a child class (as this would contradict the idea behind generalization), values of inherited attributes can be modified. (An attribute like *mass* shows why these modifications should be possible.) When a class has been associated with a personal attribute and one of its parents is associated with the same attribute later on, the personal attribute will be overridden and become an inherited attribute instead. The question that arises now is what should happen to the value that was set for the personal attribute. The same question arises when an inherited attribute value is changed, and the value of that same attribute in the parent is modified later on. Should the inherited attribute value be updated with the value of the parent? Sometimes, this is what the designer wants, but most of the time, this is not. Therefore, in case of conflicting values, user intervention is required, enabling the designer to choose whether modified inherited values should be updated or not. This can be illustrated by a *physical object* class with a *mass* of 100 kg, having a *car* as a child that inherits this *mass*. When the *mass* of the *car* is then changed to 1500 kg, and the *mass* of the *physical object* to 300 kg afterwards, it is unsure whether the *mass* of the car should be 300 or 1500 kg.

Parent-child relations are not limited to classes only. Also materials and actions can be ordered. If, for example, the *wood* material has been associated with the hardness attribute, all children of that material, such as *bamboo* and *oak wood*, inherit this attribute. For the *speak* action, possible children are *chatter*, *shout*, and *whisper*, all having the *loudness* attribute. As attributes and states are mostly unrelated, generalization is not applied to them. Units neither have parent-child relations, but they are categorized instead, as was discussed in Subsection 4.1.1.

Besides parent-child relations, the Library Editor supports another way to let (generic and game-specific) classes inherit services. This is done by introducing the *class category* component:

- *Class category*: a collection of classes that meet specific class or attribute conditions

With a class category component, it is possible to group classes that do (or should) not have parent-child relationships, but should have common behavior. A door and a chest are good examples. It should be possible to open both when they are unlocked, but as they do not have a common parent that can support this behavior with a service, separate (but identical) services should be designed for both classes. To prevent this, a class category can be created instead. For a class category, class and attribute conditions can be defined, indicating which classes belong to this category. To meet a class condition, a class should be equal to the class that is the condition, or have it as a parent. For example, when defining the *door* class as a condition, all classes with a *door* as parent will be in the category. An attribute condition is similar to a precondition, only allowing classes that have the same attribute as the

precondition, and optionally having the correct attribute value. For instance, an attribute condition might be to have a *locked/unlocked* attribute. When combined with the previously defined class condition, only *door* classes with a *locked/unlocked* attribute will be in the category. For a class category, one or more services can be defined, which are then inherited by the classes in that category. When a particular class does not meet the conditions of a class category anymore (as its parents or attributes might be changed later on), the inherited service will be removed from it.

6.3 Level Editor

Once a game class library has been created in the Library Editor, it is possible to switch to the Level Editor of the instantiation phase to design the game world and populate it with instances of the designed game classes. As for functionality is concerned, the Level Editor is mainly a stripped version of the Library Editor, because it lacks (an overview of) the generic libraries. An existing game class library should be loaded, which serves as the foundation for the instance library that should be created next (or loaded, in case one was already created earlier). An instance library stores all the information about the instances in a specific game world or level. When a game class is selected in the editor, it is possible to create an instance of it. If wanted, more instances of the same game class can be created as well, as that is what instances are for. Instances will inherit the semantic behavior of the game class, after which the level designer has the possibility to customize all attribute and service-specific values even further, as was discussed in Section 4.6. Generic libraries are absent because it is not possible to define new semantic relations (e.g. with attributes) for instances. Neither is it possible to define new services. In case inventory items were defined for a game class, instances for those items will be automatically created when the instance is created. As inventory items are always game-specific classes, it suffices to just create an instance of it, or multiple, dependent on the defined quantity. If desired, the level designer can add other instances to an inventory as well. For that, instances that were already in another inventory will be removed from that inventory, as they can only be in one inventory at the same time. When parts have been defined for a game class, something similar happens.

In addition to these features, the Level Editor contains a game window, which actually takes up the most space in the graphical user interface. In this window, it is possible to observe and navigate through the game world. For prototype purposes, this world contains a simple skybox and terrain by default, but if desired, they can be changed. In Figure 28, a screenshot of the Level Editor is shown. When an instance is created, its model will be added to the world with default position, rotation, scale, and transparency values. The Level Editor allows the level designer to adjust these values, which will be stored inside the instance component together with the semantic information. As an instance and its inventory items are now visible in the game window, it is possible to adjust the position of the inventory with respect to the instance itself. For example, when the inventory of an *oven* instance is set just below the center of its model, all inventory items will be positioned there. It is beyond the scope of this research to account for visually overlapping items and the physical volume of the inventory and its items, to check whether they fit inside. A similar offset can be defined for the parts of an assembly, in order to, for example, correctly position the *wheels* of a *car* instance with respect to its *chassis*. Because this editor is just a simple prototype to show the semantic possibilities of instances, most game-specific features that are present in a regular level editor are absent, such as the placement of

waypoints for navigation, or a more advanced terrain editing tools. However, the editor should give a good overview about how instances with services can go hand in hand with level design.

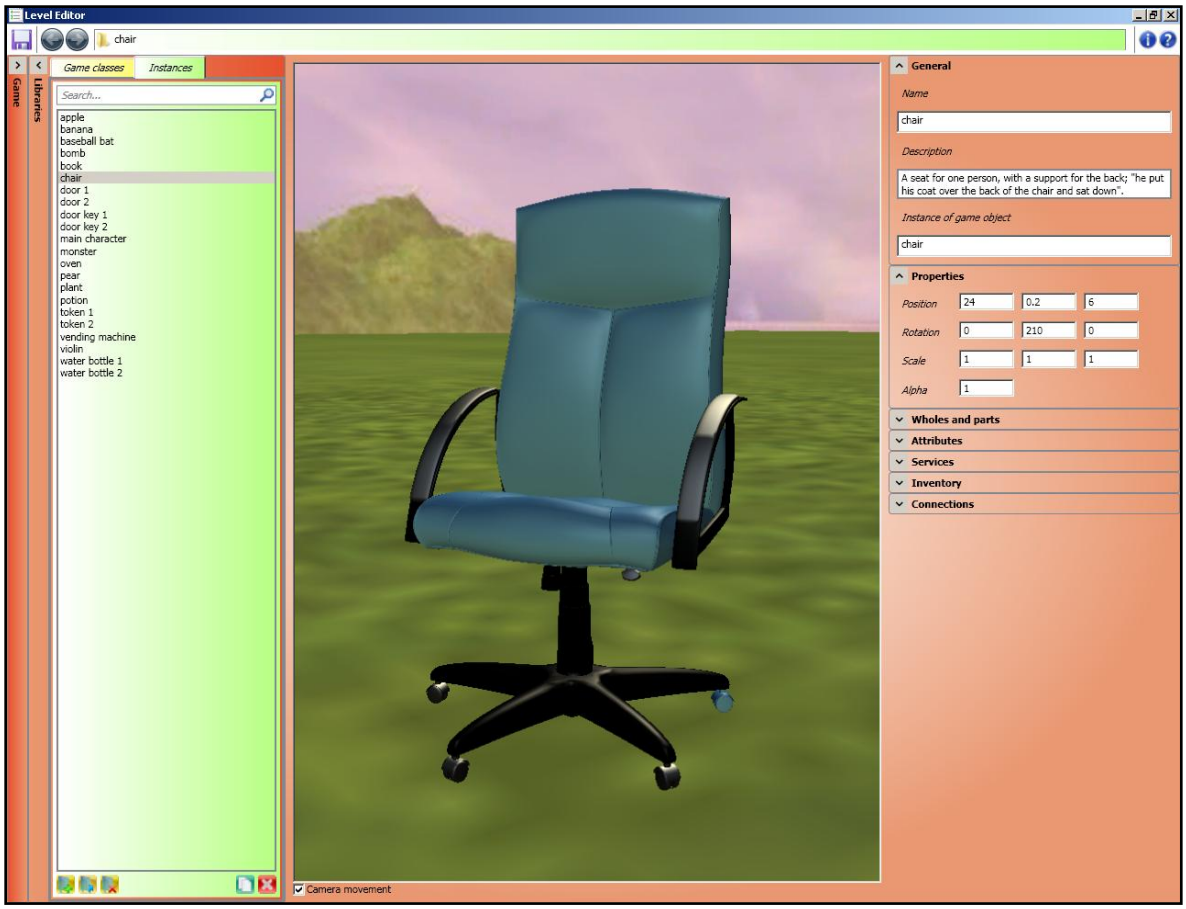

Figure 28. A screenshot of the Level Editor

6.4 Extra Features of Both Editors

In this section, several elements are discussed that are applicable to both the Library Editor and the Level Editor, such as the user interface features. Although most elements are unrelated, they have all been implemented in order to increase the usability of the editors.

In the color version of this thesis one might have noticed that each component in Chapters 4 and 5 was given its own color. This made it much easier to distinguish the components from each other. In the editors, the same colors have been used, making it possible to see in a glimpse which component is selected, or which component should be dragged to a particular list to define a relation. An example is the list where attributes can be specified. The list has an orange background to indicate that it should be filled with attribute components, which have been assigned the color orange.

Because the libraries contain many nouns and verbs from the WordNet database, they are huge. (Some numbers: more than 600 units, 6600 attributes, 13,000 actions, and 33,000 classes.) In order to find a

particular component in the editors, components can be listed alphabetically. For components that have parents and/or children, a hierarchical tree view is available as well. Starting from the root node, one can 'walk down' the tree to find the component/node one is looking for. To make searching for components even easier, a search query can be started by entering (parts of) the name in a search box, after which all possible components are displayed and the correct one can be selected. To get a quick glimpse of a component in a library, its description can be viewed when hovering over it. As an extra feature, helpful tooltips are available for all GUI elements, indicating their purpose.

For the spatial properties of service actions, an equation can be defined for the change of effect values over a particular radius. (Remember the example of the warmth of a heater.) For temporal properties, the change over time could be defined. Formulating such an equation from scratch might be hard. Consider someone who knows more or less how the effect value should change over space, but does not know the corresponding equation. On the other hand, one might know an existing equation (e.g. from physics), but might not be aware of the exact change in effect value. Visualizing the equation, as was done in Figure 21, is much easier, as one is able to directly see the outcome. To support this visualization, a Graph Plotter tool has been created, which is shown in Figure 29. From a list of predefined equations (such as power and exponential functions), the game designer can select an equation that serves as a base for his desired equation. Equation parameters can then be modified, after which the result will be plotted. If desired, the designer can customize the equation even further afterwards, and repeat this process until he is satisfied.

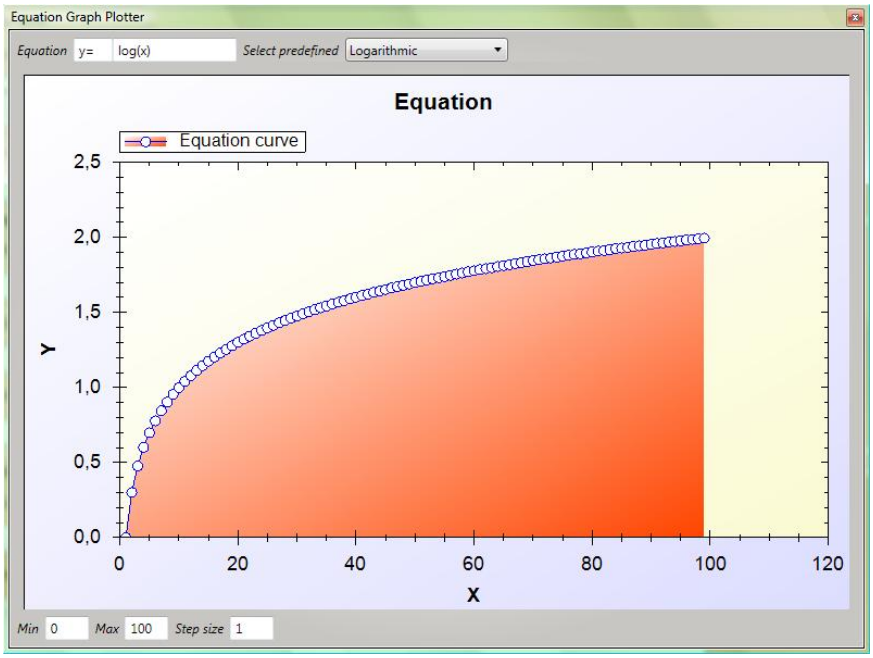

Figure 29. A screenshot of the Graph Plotter

Another useful tool that has been created is the Flow Diagram. With this tool, one is able to order the requirements and actions of a service, similar to Table 1. This results in two separate sequences of interaction steps, indicating the order in which requirements should be satisfied, and in which order

actions should be performed. A screenshot can be seen in Figure 30, in which the two sequences of interaction steps for making *coffee* is shown. *Coffee beans* and *water* are required first, after which the *coffee machine* should be *switched on*. When these requirements have been satisfied, the *machine* will immediately supply *coffee* (as there are no other actions). When no sequence has been defined before, all requirements/actions in the diagram will be positioned at the first step by default, after which they can be dragged to another step. Dropping them in between two steps is possible as well, assuming that there are at least two steps. It should be obvious that the maximum number of interaction steps is limited to the number of requirements/actions, and that requirements cannot be dragged to action steps and vice versa.

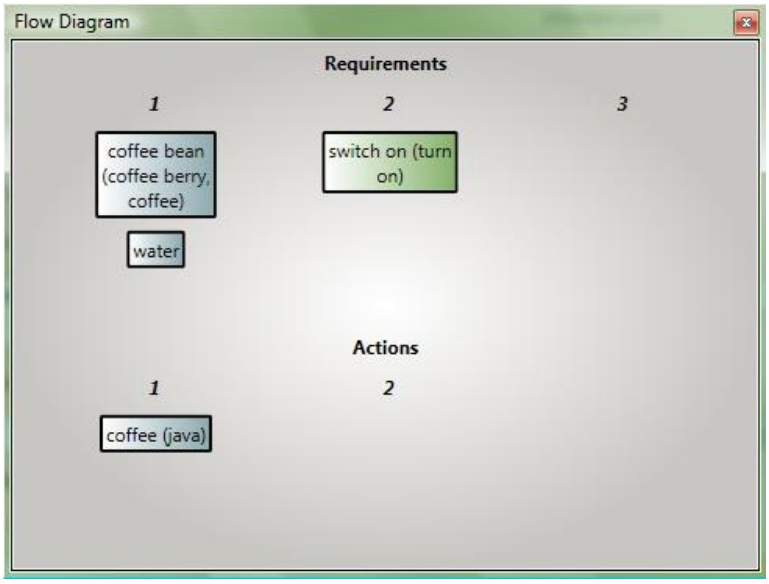

Figure 30. A screenshot of the Flow Diagram

6.5 Implementation Details

In this section, several implementation details of both editors are discussed. In-depth details are not provided; only some noteworthy details are highlighted. In Appendix A, a description can be found for each class (a C# class, that is) that has been implemented. Note that in the appendix, short descriptions for the Semantics Engine and Tech Demo are included as well, even though they will not be discussed until the next chapter.

6.5.1 Development Tools

Both editors, which only run on Windows PCs, have been implemented using the Microsoft Visual Studio 2008 (Microsoft (a), 2009) development environment. Due to familiarity with the flexible and easy-to-use C# (3.0) language, based on the .Net Framework 3.5, this was the main programming language for the system. It provided all the resources to implement the object-oriented structure of the proposed research components, to create fast algorithms to search through the libraries, and to manage relations between components. With the use of events, components could be notified of modifications of related components.

In order to design a user-friendly interface, the Windows Presentation Foundation (WPF) designer was used, which is integrated in Visual Studio. This tool allowed, among others, the design of powerful list boxes with expandable list box items and easy-to-use drag and drop functionality. Another advantage of WPF is the use of the declarative Extensible Application Markup Language (XAML) for automatic layout management, and the split between user interface implementation and the remainder of the system's implementation. Furthermore, data binding made it possible to link UI elements to properties of implemented C# classes, therefore reducing the amount of abundant methods.

For the game world window in the Level Editor, XNA (XNA's Not Acronymed) was used (Microsoft (b), 2009). This set of tools for game development could be easily integrated into Visual Studio, and was helpful to display the models of instances in the game world, together with the skybox and the terrain.

6.5.2 Populating the Generic Libraries

Section 3.3 raised the question who should be the one to populate the generic libraries and relate all their elements. Although the suggestions of that section are still valid for the second part of the question, a solution has been found for the first part. Moreover, the solution that will be proposed here has already been implemented in the generic libraries of the Library Editor. Populating the libraries was done by using WordNet, the dictionary that was used earlier to come up with definitions for some components. WordNet is a lexical library for the English language that contains many nouns and verbs, making it extremely useful for many possible classes, attributes, actions, etc. In WordNet's underlying database, nouns, verbs, adjectives, and adverbs are linked to sets of synonyms that are linked through semantic relations. This is what defines word definitions. WordNet makes use of word form and sense pairs, where the form is a string over a finite alphabet, and the sense an element from a given set of meanings. For the English language, having more than 118,000 word forms and 90,000 different senses, this results in more than 166,000 word form-sense pairs (Miller, 1995). WordNet uses the following semantic relations:

- *synonymy* (same name): words that are written different from each other, but have the same meaning. Example: a car is the same as an auto;
- *antonymy* (opposing-name): words that are opposites of each other. Example: hot vs. cold;
- *hyponymy* (sub-name) and *hypernymy* (super-name): words that have a transitive relation, similar to the generalization ontology. Example: a couch is a kind of seat;
- *meronymy* (part-name) and *holonymy* (whole-name): words that represent parts or assemblies, just like aggregation. Example: a wheel is a part of a car;
- *troponymy* (manner-name): this is like hyponymy, but then applied to verbs, which defines a transitive relation. Example: whispering is a way to speak;
- *entailment*: verbs that entail other verbs. Example: divorce entails marry.

The verbs of WordNet were used to fill the action library, as each verb represents a particular action. From the semantic relations, troponymy was then used to define the parent-child relations between these actions. As the entailment relation does not add any useful information to the actions in this research, it was not used. To populate the class library, many nouns were used. Because of the hyponymy and hypernymy relations, the correct nouns could be selected for the classes and parent-

child relations could be established automatically afterwards. Selecting the correct nouns was necessary, because there are more nouns in the English language than classes to represent game objects. Consider the classification of several nouns in WordNet in Figure 31 (be aware that many relations were omitted). All physical entities were useful for classes, while the abstract entities were not. However, those abstract entities were (after some filtering) useful to populate the attribute, state, material, and unit libraries. To include the synonymy relations of WordNet, a component in either library can be assigned multiple names. Furthermore, Subsection 4.1.2 discussed that all states have one or more related states. The defined antonymy relations in WordNet made it possible to automatically define these state sets. Finally, for classes, part-whole relationships were established immediately after applying the meronymy and holonymy relations.

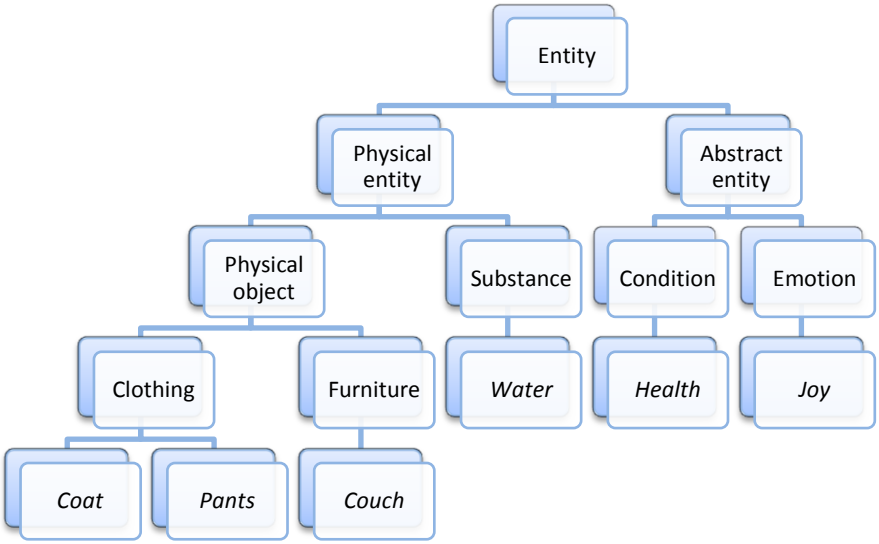

Figure 31. The classification of several nouns in WordNet

There are two final remarks about the use of the WordNet database. First, the problem of polysemy may arise, which means that a word can have multiple meanings. An example of such a word is 'keyboard', which can either be a computer keyboard, or a piano. Although this is not a problem for the implementation (as each component has its own unique identification number, which is explained in Section 6.4), it might be confusing for the game designer when he is using the Library Editor. To get around this ambiguity, synonyms can be used, but the Library Editor also makes it possible to define a description for each component. Actually, the senses of a noun or verb from WordNet were already automatically set as component descriptions. The second remark is about localization. As WordNet only contains English words, it is only useful for a Library Editor with English libraries. A direct translation to another language is not recommended, because this would definitely give problems, as one language may have more synonyms for a particular word than another language. Non-English libraries should therefore be built up from the ground. However, because English is the most common language for game development, it is unlikely that other languages are required (especially for the prototype), so this will not be a major issue.

6.5.3 Storing the Libraries

For both editors, saving and loading functionality is a prerequisite, as modifications to a component should still be there when an editor is started at a later time. A database is used for permanent storage of component information. Each time a component is modified, it is flagged. When the libraries are saved, the flagged components are updated in the database. In addition, newly created components are added to the database, while deleted components are removed from the database. When the editors are loaded at a later time, the database is loaded, by generating components from the information that is stored in it, and restoring the relations.

The database itself is similar to the one of WordNet (described in much detail in the author's Research Report), and consists of two files for each library: a data file, and a relations file. This actually makes the database a collection of files. In a data file, a component is stored under a unique identification number, together with its names and description. In a relations file, an ID entry is present again, although it is now accompanied by other IDs that indicate components to which it is related. Dependent on the type of component, values and/or quantities are stored here as well.

As an example, a small piece of the data and relations entry of a *physical entity* class can be seen in the first block below. It is related to the *mass* attribute (because the relations entry contains the ID `04965327`), of which a part of the data and relations entry can be found in the second block. The *mass* attribute itself is related to two units: *gram* and *ounce* (of which the two data entries are given in the third and fourth block), indicating that it can be expressed with either of the two. For the *physical entity* class, one has chosen to define a *mass* of `25.6` *grams*, as that value has been defined in the relation entry (next to the ID of the *mass* attribute), together with the ID of the *gram* unit.

```
DATA:  00001930 | physical_entity | An entity that has physical existence.

RELATIONS:    00001930 | 04965327 25.6 13543166
```

```
DATA:  04965327 | mass | The property of a body that causes it to have weight in a
gravitational field.

RELATIONS:    04965327 | 13543166 13539515
```

```
DATA:  13543166 | gram gramme gm g | A metric unit of weight equal to one
thousandth of a kilogram.
```

```
DATA:  13539515 | ounce oz. | A unit of weight equal to one sixteenth of a pound or
16 drams or 28.349 grams.
```

The database is not only used for the Library and Level Editor, as it will also be used for the second subsystem of the prototype. In the next two chapters, the Semantics Engine and the Tech Demo will be discussed. The latter requests the former to load the available components and their relations from the

database. The instances in the instance library are then placed inside the game world of the Tech Demo, after which the Semantics Engine takes care of all their semantics.

7 ■ Prototype – Semantics Engine

The previous chapter discussed the prototype subsystem that focuses on game development. This chapter will elaborate on the subsystem that focuses on games themselves. Even though semantic objects can be created with the Library Editor and placed in a game world with the Level Editor, there should be a way to take care of the designed services. This chapter will give a solution in the form of a Semantics Engine, which will be extensively described.

A general description of the engine will be given in Section 7.1, after which Section 7.2 will describe what happens when the engine is updated. Section 7.3 describes how the engine can check whether service requirements have been satisfied. Finally, Section 7.4 discusses how game developers can use the engine to support object interaction.

7.1 Purpose and Use of the Semantics Engine

Specifying services for game objects is useless when a game does nothing with this additional information. Saddling game programmers with the task to handle object interaction and take care of services is not the desired solution, as this would give them much more work. Furthermore, the implementation that takes care of services would be similar for each game development project. To aid game programmers, the *Semantics Engine* has been developed. This engine can be coupled to a game, allowing game programmers to use it when necessary. The Semantics Engine:

1. maintains all the instances that have been created by level designers during the instantiation phase;
2. supports the execution of actions by instances;
3. takes care of services, by checking whether requirements have been satisfied, and, if so, performing their corresponding actions;
4. offers game programmers useful methods to improve in-game object interaction.

When programmers are using the Semantics Engine in their game, most of its techniques stay behind closed doors, as they do not have to know the implementation details. Instead, programmers are provided with several methods to access the required functionality. First of all, the engine should be initialized by loading an instance library that was created in the instantiation phase. This way, the engine is made aware of all available semantic instances in the game world, making sure that all their semantics is updated when necessary. During the initialization, the game itself is also made aware of these instances, as their models should be drawn on the screen, and a programmer might want to pass them

to a physics engine. Other public and private methods will be discussed in the remainder of this chapter, as they are more complex, and thus require more attention.

7.2 Updating the Engine

To make the engine actually do something, it should be updated when the game is updated. It is up to the programmer to decide with which time interval the engine is updated. For example, this could be done each frame, or once a second. Whatever the programmer chooses, it is important that the correct elapsed time since the last update is carried along, as this is required for services with temporal properties. However, the more often the engine is updated, the more accurate and up-to-date the instances will be.

When the engine is updated, only semantic instances with active services are updated. These are the instances of which the service requirements have been satisfied, and of which the actions should be performed. The next section discusses how these instances can be determined; this section will first discuss what happens during an update. In short: for each instance with an active service, the service requirements are taken care of, after which the corresponding service actions are performed. This may result in changes of attribute values or inventory items of affected instances. As this might have had influence on their services, for these modified instances is checked whether inactive services have become active, or active services have become inactive.

Requirements

For an active service, requirements are handled first. Note that at this point, it is already known that the requirements of that service (if any) have been satisfied, as the service is already active. Therefore, there is not much to do here, except for the two following checks for passive actions and demands.

Passive actions

When a passive action has been satisfied, this means that an active action with the same action component was performed on the instance with the service. That action component might state that the target of the action should be exhausted. Think of an apple that should be exhausted when it has been eaten. If this is the case, the instance should be removed from the Semantics Engine and the game. Therefore, it is flagged, so it can be removed in the next update. It cannot be removed directly, as it might have actions of its own that should be performed first (like lowering the level of hunger of the one that eats the apple).

Demands

The second check should be made for demands. Although some demands are only used, others are exhausted, like coffee beans and water when making coffee. In this case, the demands should not only be removed from the inventory of the instance with the service, but also from the engine and the game. For this, three aspects should be taken into account: the correct quantity, whether the demand has been defined as a generic class, a game-specific class, or an instance, and whether the demand is a physical object or a substance. If the demand has been defined as a class, the engine searches the inventory for items that are based on that class. Then, the correct quantity is removed. For physical

objects, the correct number of instances is removed from the inventory, engine, and game. For substances, the correct amount is subtracted, while taking volume units into account. The substance can be completely removed when it does not have a quantity anymore (when its quantity equals 0 liters/gallons/etc.). In case a demand has been defined as a specific instance, the engine will search for that one in the inventory and remove it.

Actions

After the requirements have been taken care of, the actions of active services are performed, thereby taking the sequence of interaction steps into account. Actions in the first step are performed first, then the actions in the second step, and so on. However, before an instance is able to perform an action, the targets should be determined, as was defined in Subsection 5.2.3. When all instances in a particular radius are the targets, the engine calculates the Euclidian distance between the position of the actor and the position of all other instances, and checks whether that distance is smaller than the value of the defined radius (thereby taking possible unit of length conversions into account). On each target, the action can then be performed, which means that it is possible that an active action is performed, or a particular instance is supplied. It is, however, not certain that this will occur, as probabilities play a role as well. When an action is independent from other actions, the engine will randomly pick a value between 0 and 1, and check whether that value is lower than the defined chance of the action. If so, the action can be performed. For an action that is dependent on other actions, the engine selects only one of those actions. For this, the distribution of probabilities is taken into account, so actions with a higher chance are more likely to be selected.

Furthermore, actions that are continuous are not performed in each update loop, but only when the correct amount of time has elapsed (see the part about intervals in Subsection 5.2.4). If, for example, an interval has been defined of 1 second, and the engine is updated half a second, the action will be performed in each 2nd update step. If, however, the interval has been defined as 0.5 second and the engine is updated each second, the action will be performed twice during one update.

Active actions

For an active action, one or more effects have been defined. Each effect will try to affect an attribute of a target. Only when the target has this particular attribute, it can be affected. When the effect is defined as a state change, the target's attribute will switch to this state. For an attribute with a numerical value, that value is increased or decreased with the value that was defined at the effect of the active action. If required, the values are first converted to their base units, as it is not possible to, for example, directly add a value in *degrees Celsius* to a value in *degrees Fahrenheit*.

In case an active action is subject to spatial properties, the effect value is modified first, before adding it to, or subtracting it from, the targets attribute value. For spatial properties, the Euclidian distance between the actor and the target is calculated, after which it is passed as an argument to the function over space, together with the effect value. The result of this function is a modified effect value, which is passed to the function over time if temporal properties have also been defined. In this case, the time that has elapsed since the service became active is passed along as well. To support this, each active service keeps track of this time, which is updated during each update loop with the elapsed time

since the last update. This also results in a modified effect value, which is then finally added to, or subtracted from, the targets attribute value.

There is one remaining check that is made when active actions are performed. The Semantics Engine checks whether the target has any passive action requirements that are based on the same action component as the active action. (For instance, a *person* has the *eat* action component as active action, and an *apple* has it as a passive action.) If that is the case, the passive action is flagged as satisfied, but only if all the requirements in earlier steps of the sequence of interaction steps (see Section 5.3) have been satisfied as well. For example, when an instance has a service with three unsatisfied passive actions in three subsequent steps and an active action is performed on this instance that can satisfy the second passive action, it will not be satisfied, as the first passive action should be satisfied first.

Supplies

Besides performing active actions, an actor is also able to supply an instance to a target, which has been defined as a special action, and is taken care of by the Semantics Engine in a different way. There are several possibilities to determine which instance should be supplied, as it is based on what the designers have defined in the three design phases. The biggest difference is whether the supply should be newly created, or given from the inventory of the actor. Another issue is whether the supply has been defined as a generic class, a game-specific class, or a particular instance, because in a game, only instances can be supplied. (Note that a combination of supplying a particular instance and a newly created supply is not possible.) For example, a vending machine for candy might supply any type of *candy* (generic class), only *chocolate bars* (game-specific class), or a particular *chocolate bar* with a golden ticket (instance). Finally, there is also the difference between the type of supply: whether it is a physical object or a substance.

Supplying a new instance of a game-specific class is done by creating a new instance of that class. When the supply is defined as a generic class, the engine should first search for available game classes that are based on that generic class, pick a random one, and then create an instance of it. In case of the substance type, the instance should also be assigned the correct quantity in the correct unit, which is defined at the supply. For physical objects, the correct number of instances should be taken into account, as it is possible that more than one instance should be supplied.

When the supply should be given from the inventory, another strategy should be used. In case a specific instance should be supplied, the inventory will be searched for this instance. If found, it will be given to the target; otherwise, it is bad luck for the target. If there are multiple targets, only the first one will receive that instance. In case the supply has been defined as a class, the engine searches for an instance in the inventory that is based on that class. For physical objects, the engine tries to supply the correct number of instances from the inventory, as long as the inventory contains the correct instances. For substances, the engine tries to supply the correct amount of an instance. An example is the following: suppose that an actor has 2 *liters water* in its inventory and 0.5 *liters* should be supplied. The engine will then split the instance of 2 *liters* in two instances, where the instance with 1.5 *liters* remains in the inventory, and the instance with 0.5 *liters* is supplied to the target.

When an instance is given to a target, it is added to the inventory of the target. In case the gift is a substance and the target already got an instance in its inventory that is based on the same game class,

both instances are mixed into one instance. The quantity of the resulting instance will, of course, be the sum of both quantities, thereby taking units (and their conversion equations) into account. If, for example, 1 *liter water* is given to an instance with 2 *gallons*, both values are converted to their common base unit, added together, and then converted back to the original unit, thus resulting in 2.26 *gallons*.

Similar to active actions, an additional check is made at the target to find out whether the supply corresponds to a demand of one of the target's services. If so, and if the demand is one that should always be given (instead of only being available in the target's inventory; see Section 5.3), and if the previous requirements in the sequence of interaction steps have been satisfied, the demand will be flagged as satisfied.

7.3 Checking for Satisfied Requirements and Active Services

When an action has been performed by an actor on a target, both instances may be modified (either a change of an attribute value, or the addition of, or removal from, an item in the inventory). For the actor, this means that the service that has been active thus far might not be active anymore. For the target, this means that a service of which the requirements were not satisfied thus far, might be satisfied now, resulting in an active service of which the actions should be performed in the next update. Therefore, the services of the actor and the target are checked after an action. For this, preconditions are checked first. Then, it is checked whether passive actions, demands, and part requirements have been satisfied.

Preconditions

Preconditions of an instance are compared to the attributes of that instance. For an attribute that is defined as a state, it suffices to check whether the state corresponds to the state of the precondition. Dependent on the inequality statement of the precondition, the precondition will be satisfied or not. An attribute with a value is compared to the value of the precondition, taking the inequality sign into account. In case units are defined as well, the values have to be converted to the base units first, before they can be compared. When the base units are not the same (although the current implementation of the Library Editor prevents this), the precondition is immediately marked as not satisfied. In case an AND/OR relation has been defined, an additional second check is made for the other inequality statement.

Passive actions

At this point, passive actions already know whether they are satisfied or not. After all, the engine immediately checks whether performed active actions satisfy any passive action of the target. Therefore, it is only important here to check whether all passive actions have been satisfied.

Demands

Checking whether a demand has been satisfied is similar to supplying an instance from the inventory. When a specific instance is demanded, the Semantics Engine searches for this instance in the inventory of the actor. When the demand has been defined as a class instead, the engine searches through the

inventory to find an instance that is based on that class. For this, quantities are taken into account. Only when the available quantity is greater than, or equal to, the demanded quantity, the demand may have been satisfied. For substances, the engine searches for the correct amount of the substance. What is left to check are the optional preconditions of the demand, which is similar to the precondition checking that was described above. However, in this case, preconditions are compared to the attributes of the found demand in the inventory, and not to the attributes of the actor.

Part requirements

Check whether part requirements have been satisfied depends on the type of requirement. In case a part requirement states that a service of a particular part should be satisfied, it suffices to check whether that is the case, by checking the requirements of that service, just like above. In the case the requirement is to have all parts attached, the parts of the instance are counted. If that number corresponds to the default number of parts of the game-specific class on which the instance is based, the part requirement has been satisfied. An example is the *car* game class, for which four *wheels* have been defined as parts. Before being able to drive, the car should have all its four *wheels*, which could be defined as a part requirement. When an instance of that class only has three *wheel* instances as parts, the requirement is thus not satisfied.

Active services

When all requirements of a service have been satisfied, the service will be marked as active, indicating that its actions will be performed in the next update step. However, dependent on the duration of an action (see Subsection 5.2.4), it might not be performed anymore at some point. If this is the case for all actions of a service, the service will become inactive, meaning that it will not be updated anymore (at least, until it becomes active again). Checking whether an action should still be performed is first done by looking at the type of the action. If the action is discrete, the action should be performed the number of times that is defined at the frequency. Each time the action is performed, a counter is increased. When the counter has reached the frequency, the action will not be performed anymore in subsequent update steps. For continuous actions, there are more possibilities. An infinite action will never stop, meaning that the corresponding service will remain active as long as the instance with that service exists. An action that should be performed while the requirements are satisfied will not be performed anymore in case this no longer applies, as an attribute value or inventory item may have been changed, resulting in a precondition that is no longer satisfied, or a demand that is no longer available. For actions that should be performed for a fixed amount of time, a timer is increased each update with the elapsed time since the last update. When the timer exceeds the specified time, the action will not be performed anymore in the next update. Finally, it is possible that actions should be stopped by an intelligent entity. This, however, requires more interaction than was discussed thus far, and is further discussed in the next section.

7.4 Supporting User Interaction

Up to this point, the most important aspects of the Semantics Engine have been discussed. However, all interactions between the instances in the engine (the handling of services) were done on autopilot. If,

for example, the *sun* was designed as an entity that continuously heats up the area, the engine will increase the *temperature* of surrounding instances correctly, possibly satisfying one of their preconditions. There is, however, a variable aspect that should not be forgotten: besides static instances, the world can contain dynamic, self-thinking, instances as well. The most obvious one is an avatar that is controlled by the player. Others are computer-controlled humans, animals, aliens, etc. These instances do not have to automatically perform the active actions of their services when the corresponding requirements have been satisfied. Instead, they should have the ability to choose of which service they want to perform the actions, and when. To support this in the Library Editor, an option has been added to a service to indicate whether the actions of a satisfied service should be performed *automatically*, or *on request*. To support the latter in a game, user interaction is required. Partially, this should be done by game programmers themselves, as they should specify gamepad buttons, keyboard keys, mouse buttons, or selection menus (to let the player select an action to perform), in addition to the implementation of navigation controls (to make sure that a player can navigate its avatar through the world), and visuals (to show the avatar and its inventory). Partially, this can be done by the Semantics Engine, as it provides game programmers some public methods that can be coupled to their own implementation. These methods, such as requesting which actions are possible on a specific instance, or actually perform an action, will be discussed in the remainder of this section.

Actions that are related to physical movements (walking, running, jumping, etc.) are not entirely supported by the Semantics Engine, and should still be implemented by game programmers. Due to the complexity and variation of these actions (which are often bound to a physics engine), it is better to let programmers take care of them. What could be done, though, is to define attributes for a game class that represents the avatar of the player. Possible attributes are the *movement speed* and *running speed*, which can then be used in the game when avatar movement is implemented. For that game class, a designer could also define a service with the *run* action, and indicate that the corresponding 3D model should play a specific animation when it is performed. Furthermore, another service might state that when the avatar is running, its *fatigue level* decreases, which can be defined as another attribute.

One of the features of the engine is to request which actions an instance can perform at a particular moment. For this, the engine checks for each service of that instance whether it has any active actions. If so, it then checks whether the requirements of the service are satisfied. If that is also the case, the active actions are marked as actions that can be performed by the instance. As an extension to this, another feature is to request all the actions that an instance can perform on a particular target instance. To accomplish this, the engine checks whether the previously found active actions have effects that affect any attribute of the target (for example, a *person* can only open entities that have the *opened/closed* attribute), or whether the target has one of those actions as passive actions (for example, the *eat* action of a *person* can only be performed on entities with *eat* as a passive action). If there is a match, the active action is valid. To extend this feature even further, another option is to search for useful actions only, which are actions that are geared towards satisfying a service of the target. In this case, active actions are valid when they satisfy a passive action of the target, or when they affect a precondition of the target in a positive way. For example, when a heater is in the *off* state, and it has a service to heat the area when it is in the *on* state, the *turn on* action is useful. These extra features

are not limited to regular actions, as it is also possible to request the demands and supplies of an instance, and whether it is useful for an instance to give a particular inventory item to another instance. Besides requesting individual actions that can be performed, there is also an easier, top-down, approach, in which the engine allows the programmer to request all services with satisfied requirements, after which the engine can perform all the actions of one of those services simultaneously.

Instead of coupling these methods to the implementation of the player only, artificial agents can also be programmed in such a way that they make use of them. With backward chaining, an artificial agent might be able to make use of instances with services. For example, suppose that an agent has a low *temperature*. Therefore, it could search for instances with the *heat* action, as that action raises the *temperature*. The engine can help it to tell which instances have this action, and which requirements should be satisfied in order to trigger it. This way, an agent can find out that he is able to stand near a *heater* and *turn* it *on*. Programming the behavior of artificial agents, however, is a task for AI programmers, and is therefore not further covered in this research.

Although game programmers can request the engine for (useful) actions of an instance (on a target), it is up to the developers to decide how the player is presented with these actions. However, when a particular action should be performed, the Semantics Engine can be consulted again, by letting it know that a particular instance wants to perform a particular action, possibly on a particular target. Instead of immediately performing the action, the engine waits until the next update step, so that all actions of all instances are taken care of at the same moment. Executing an action is not done immediately, as this might result in infinite loops of the engine, which is then most likely due to mistakes of the game designer in the customization phase. Consider two *person* entities with a service that they hit back when they are hit by someone. Then, if person A hits B, B will hit back, which will make A hit B again, and so on, lasting forever. When only one action is executed per update step, this hitting process will still be repeated infinitely, but not at the cost of the engine and the game.

The *give/supply* action of the Semantics Engine is also accessible to the programmer. When coupled to a game, an instance, such as the avatar, is able to give an instance from its inventory to another instance. A player could, for example, accomplish this by standing with his avatar near another entity, browse through his inventory, and select an item to give away. A common game example is to put unused items into a storage chest. A similar action that is supported by the engine is to *discard* unwanted inventory items. Instead of being put in another inventory, an item will be left on the position where it was discarded, possibly subject to physics afterwards, if the game itself supports this. The engine also supports the opposite of this action: the *pickup* action. This action is commonly found in games: think of picking up health packs, weapons, keys, or food from the ground, or picking up an item that was stored in a container. In a game, an instance, such as the avatar, might select another instance and perform the *pickup* action on it, after which the engine actually adds the item to the inventory. Of course, all the previous actions might satisfy the demands of an instance's service.

Besides an inventory, this research also introduced the notion of parts and wholes. Up to this point, parts could be defined for an entity that represents an assembly, but there was not much that showed their practical use, except for the inherited attributes and requirements of a service. Parts and

wholes, however, also introduce more interaction possibilities. Therefore, the Semantics Engine supports two more special actions: *attach* and *detach*. This way, a player is able to pick up an object and attach it to an object that represents an assembly. Of course, this should not be possible when the assembly does already have all its parts, or when this part has not been defined for the assembly. The *detach* action is the reverse of the *attach* action, and should not require extra explanation. With these actions, puzzle elements can be introduced to a game, where, for example, missing parts of a machine should be found, before the machine can be operated.

In the Library Editor, it is possible to define, for example, that a *key* can unlock a *door* (by defining a service for the *key* class with the *unlock* action as an active action). However, the *key* will not magically present itself to the *door* to open it. Instead, a *person*, such as the avatar of the player, should use the *key* on the *door*, after which the *key* will *unlock* the *door*. This is seen often in games. To support this functionality in the engine, the *use* action has been defined. This way, an in-game instance can use an instance from its inventory on a target instance. In case the requirements of the inventory item have been satisfied (for example, a precondition of the key might be that it is not broken), its actions will be performed on the target.

At this point, all features of the Semantics Engine have been discussed. The next chapter discusses how this engine is put to good use in the Tech Demo, which also includes the user interaction methods that were described in this section.

8. Prototype – Tech Demo

The Library Editor was introduced to design semantic classes, the Level Editor to place instances of those classes in a game world, and the Semantics Engine to handle the instance's services. It is, however, still a bit unsure whether these tools actually help in achieving the research goals. Theoretically, object semantics can easily be improved with them, but in practice, it is unknown whether the gameplay is improved when more object interaction is available. In the end, it is up to game designers to create a good game. Due to time constraints and resource limitations, it was not possible to produce a full-blown game that optimally makes use of services. What was possible, however, was to develop a simple Tech Demo for the PC that demonstrates the potential of the approach in this research. Furthermore, it was a perfect way to test the Semantics Engine. Just like the game world window of the Level Editor, the Tech Demo has been created with XNA.

In first-person view, the user/player is able to walk around with his avatar in an environment that is filled with semantic instances. There is no actual gameplay involved, and the player has no goals to achieve; the world is just a small playground to test the services of the placed instances. Although the avatar itself is invisible for the player, it is a semantic entity: an instance that has been created from a game-specific *person* class, which, in turn, has been based on the generic *person* class. For this class, several services have been defined to indicate which (active) actions a *person* can perform. Some of these actions are *open* and *close*, *turn on* and *turn off*, *sit*, *hit*, *eat*, and *read*. As the avatar is based on this class, the player can make his avatar perform these actions. Furthermore, the special actions of the Semantics Engine are available to the player. This way, it is possible to, among others, *pick up* instances that are lying around in the world, after which they are added the avatar's (initial empty) inventory. Then, they can be *supplied* to other instances. Several attributes (like *health* and *level of hunger*) have also been associated with the *person* class, so the avatar can be affected by the actions of other entities. In the Library Editor, the generic *person* class has been specified with these generic services and attributes, while specific values have been customized for the game-specific *person* class. In the Level Editor, the person instance was then given an initial position and rotation in the world. During the implementation of the Tech Demo, the avatar was extended with movement and orientation controls (making use of the keyboard and mouse). Besides the avatar, other entities have been designed as well. Some of them will be described in the next chapter.

In the Tech Demo, two different modes of play have been implemented. This also shows the flexibility of the Semantics Engine, as each mode gives the player a different playing experience.

1. The *Request mode* is similar to how the game *The Sims* works. The player is able to navigate his avatar to an instance in the world and select it to request the actions that can be performed by the avatar on that instance. The resulting list of actions is the logical intersection between the active actions of the avatar and the passive actions of the target. The list is further filled with active actions that can change one or more attributes of the target. The player can then select one of these actions, after which his avatar performs it. When this action satisfies the requirements of one of the target's services, the target might perform its own active actions as a response. On one hand, this mode is easy to use and easy to understand. After all, all possible actions on an instance are offered to the player when he selects that instance. This way, the player cannot make 'mistakes', as he is certain that each performed action affects the target in some way. On the other hand, the mode is not completely realistic, as an entity in the real world does not 'tell' others which actions can be performed on it. By using common knowledge, a human knows which actions are useful on a specific target, and which ones are not. Therefore, this mode is ideal for simpler games, and especially helpful for artificial agents.
2. The *Free mode* is more based on the real world and is in some way similar to old text-based adventure games like *Zork* (Infocom, 1980). In this mode, the player is able to make his avatar perform each active action that has been defined for it. Of course, this is only possible when the requirements of the corresponding services have been satisfied. Having the possibility to perform each action does not automatically mean that it will be successful. Whether the action is successful or not, the avatar might be affected by its own action. If the action is performed on a particular target, the target checks whether this satisfied one of its services, and possibly performs another action in return, just like in the Request mode. An example in this mode is the *open* action. If the player wants, he can perform this action anytime and anywhere. So if he performs the action on a *table*, this is possible, although it will not have any effect. When performed on a *closed door*, however, something will happen, as for the *door*, a service has been defined which states that it will open when the open action is performed on it.

Although the Tech Demo demonstrates two modes, a real game that makes use of services would probably just use one (assuming that a game development team wants to develop a game with improved object interaction). Note that a game development team is not required to exactly adopt one of the given modes, as they were just two possible examples. With some creativity, developers can add a twist to one of the modes, in order to adjust it to their own game. However, whatever twist is added, in the end it will probably be similar to one of the given modes, as this is what the current implementation of the Semantics Engine offers. Dependent on the desired level of complexity, or to what extent object interaction from the real world should be imitated, a development team might choose to either follow the Request mode or the Free mode. For more realistic object behavior, the latter mode is the mode to use. For some games, however, this might not be the best choice, as it unnecessarily increases the complexity, especially when there are a lot of actions that can be performed. After all, there should also be a way for a player to select an action. A player might be presented with a list of actions to choose from, but actions can also be coupled to keyboard keys or

gamepad buttons. For the last case, the number of buttons, and therefore the number of possible actions, is limited, which might be a problem when an avatar can perform many actions (consider all the actions that a person can perform in real life). With the recent technologies in motion controls (like the *Wii remote* for the Wii (Nintendo, 2009), or *Project Natal* for the Xbox 360 (Microsoft (c), 2009)), new pathways are being opened for players to convert their physical movements to in-game actions, allowing the selection of many actions in a way that also feels more natural. These technologies, however, are still in infancy stage, and it is up to developers to choose whether they suit their game. For both modes, the Tech Demo makes use of a list of possible actions that is shown on-screen, through which the player can navigate with the mouse wheel, and of which an action can be selected by a click of the mouse.

9. Scenarios and Use Cases

Chapters 3, 4, and 5 presented a three-phased game development process in which components and services can be specified. The three chapters after that elaborated that design into a prototype. As each chapter discussed a particular topic, it might still be unclear to get the whole picture. Therefore, this chapter will shine a light on the complete design process of a virtual entity, and give more insight into components and services by giving several fictional scenarios and use cases. Readers who have no interest in these additional examples can skip this chapter and jump directly to the conclusions.

Each scenario will show how a service can be used to quickly define object behavior. Some scenarios are based on interactive objects in current games, while others will show the services of some less common game objects. The first scenario of this chapter starts with the specification of a generic class. By going through the subsystems of the prototype phase by phase, described in multiple use cases, this class will eventually 'evolve' into a semantic instance in a game world. The majority of the remaining scenarios, however, will combine the first two phases in one use case and omit the third phase, as most events will be similar to those in the first scenario. In this chapter, it is assumed that the libraries in the specification phase have already been populated, but their elements have not yet been related. Furthermore, unless defined otherwise, the actor will always be a game designer.

Section 5.3 pointed out that it is possible to make a distinction between several service types. This chapter is structured in such a way that three of them have been given their own section with scenarios and use cases. In Section 9.1, two scenarios will be given of entities whose services are of the 'exchange' type. Section 9.2 will then show three scenarios of the 'act' type. Finally, two 'mixture' scenarios are described in Section 9.3.

9.1 Exchange

In an exchange, one or more demands are exchanged for one or more supplies. These supplies can be given from the inventory, or they can be newly created. This section will give an example of both.

Ticket machine

Goal: Design a ticket machine to buy a train ticket.

Scenario: Niko, the game designer of *Grand Theft Auto V*, wants to add a ticket machine to all stations in Vice City, from which the player can buy a ticket for the train. (Otherwise, he risks getting himself a fine of the police.) Therefore, Niko wants to design a ticket machine that can be placed inside the game world by his level designer Tommy.

Use case 1: Design a generic *ticket machine* class.
Actor: Game designer Niko.
Events: Niko:
- creates a new generic class;
- changes the default name to 'ticket machine', and sets a description;
- adds a new service to the class, and selects the 'exchange' type;
- drags the *coin* class to the list with demands;
- drags the *ticket* class to the list with supplies.

Use case 2: Design a game-specific *ticket machine* class.
Actor: Game designer Niko.
Prerequisites:
- The generic *ticket machine* class has been designed;
- An artist has created a 3D model of the machine;
- A game-specific class library has been created or loaded.

Events: Niko:
- creates three new game-specific classes with names 'coin', 'ticket', and 'ticket machine';
- drags the *coin*, *ticket* and *ticket machine* classes to the corresponding game classes;
- selects the inherited service of the *ticket machine* and selects the created *coin* game class for the demand, and the created *ticket* game class for the supply;
- sets the quantity of the demand to 1 and selects 'give always' (to indicate that a coin should always be given in order to get a ticket);
- drags the *ticket* class to the inventory of the *ticket machine* class, and sets the quantity to 1000;
- sets the quantity of the supply to 1, selects 'from inventory', and sets the spatial property to 'trigger' (to indicate that 1 ticket should be supplied from the inventory to the one who triggers this service).

Use case 3: Create a ticket machine instance and place it in a game.
Actor: Level designer Tommy.
Prerequisites:
- The game-specific *ticket machine*, *coin*, and *ticket* classes have been designed;
- An instance library is loaded, together with the game classes library containing these classes.

Events: Tommy:
- selects the *ticket machine* game class and creates an instance of it;
- changes the X, Y, and Z values of the position, rotation, and scale factor.

Notes: To use the *ticket machine* instance in the game, the avatar of the player (an instance that can be based on a *person* game class that, in turn, can be based on the generic *person* class) should have at least one *coin* instance in its inventory, which can be given to the *ticket machine*.

Related scenarios: Shops in building simulations (like *Rollercoaster Tycoon*), stores in role-playing games (like *Dungeon Siege*), vendors in action-adventures (like *Resident Evil*), or candy, soda, and weapon vending machines in first person shooters (like *Doom* and *BioShock*). Even quest givers in (massively multiplayer online) role-playing games (like *Guild Wars*) are

related, which request the player to find a specific weapon/potion/scroll, in return for money.

Weapons factory

Goal: Design a weapons factory to produce tanks.

Scenario: Unsurprisingly, the weapons factory makes its return in *Command & Conquer 4*. It is up to game designer Yuri to design this factory. Each time the player provides the factory with enough steel, it will produce a tank, which can be used to bomb the enemy.

Events: Yuri:

- creates *weapons factory*, *steel*, and *tank* game classes, and bases them on the corresponding generic classes;
- adds a service of the 'exchange' type to the *factory* game class, giving it the name 'produce tank';
- selects the *steel* as a demand with a quantity of 5 megagram (= 5000 kilogram);
- defines that the demand should be available in the inventory, and that it is exhausted when the service becomes active (so when the inventory actually contains 5 megagrams of steel);
- selects the *tank* as a newly created supply with quantity 1;
- indicates that the service actions should be performed 'on request'.

Notes: When the weapons factory should be able to produce more weapons than tanks only, a designer can create more services, each having other requirements and another supply. To allow the player to choose between the services, one of the programmers can request these services from the Semantics Engine, and present them to the player.

Related scenarios: Any other production buildings, like (saw) mills or coal mines in real-time strategy games (like *Age of Empires*) and building simulations (like *Anno 1602*).

9.2 Act

This section will give three scenarios of an act, which is a service with a particular active action, possibly subject to a passive action requirement.

Door and key

Goal: Design a key that only unlocks one specific door.

Scenario: For *BioShock 3*, game designer Andrew and level designer Frank are planning to add a prison to Rapture with cell doors that can only be unlocked by specific keys. Before a player can open a door to a prison cell, he first has to find the correct key, as other keys do not fit the lock. Andrew wants to design two door and key combinations.

Events: Andrew:

- creates a *locked/unlocked* attribute (remember from Section 4.1.2 that there is no perfectly suitable attribute for some states);
- adds the states *locked* and *unlocked* to this attribute and assigns it to the generic *door* class, setting the default state to *locked;*
- defines the *unlock* action component as discrete, and adds the previous attribute as an effect that changes the state to *unlocked*;

- adds a service of the 'act' type to the generic *key* class, and sets the *unlock* action as an active action;
- selects 'specific target' for the spatial properties of the active action;
- creates game-specific classes for the *door* and *key*, based on the generic classes;

Frank:

- creates instances of those game classes: *door 1*, *door 2*, *key 1*, and *key 2*;
- defines the *door 1* instance as the specific target of the inherited *unlock* active action of the *key 1* instance;
- does the same for *key 2*, but now with *door 2* as the specific target.

Notes: To create even more door and key combinations, it suffices to create more instances of the *door* and *key* game classes, and change for each *key* instance the specific target of the *unlock* action. To make sure the *open* action can only be performed on the *door* when it is *unlocked*, a service of the 'passive action with precondition' type can be defined (no section in this chapter has been dedicated to this type). This service would then have the *open* action as a passive action, and a precondition for the *locked/unlocked* attribute, requiring that it should have the *unlocked* state.

Related scenarios: Doors and keys in puzzle games (like *Myst*), and chests and keys in role-playing games.

Apple

Goal: Design a delicious apple that can be eaten.

Scenario: In *The Sims 4*, it is possible to grow an apple tree, and Sims are able to pluck the apples. Game designer Will would also like to see that Sims are able to eat those apples, in order to satisfy their hunger.

Prerequisites:

- A generic *person* class has been created;
- The *person* class has a service with the *eat* action component as an active action;
- The *person* class has the dimensionless *level of hunger* attribute, with a default value of 50, minimum value of 0, and maximum value of 100.

Events: Will:

- defines the *eat* action component as a discrete action that exhausts the target;
- defines the *satisfy hunger* action component as a discrete action with the effect of decreasing the *level of hunger* attribute;
- adds a service of the 'act' type to the *apple* class with the *eat* action component as passive action;
- assigns the *satisfy hunger* action component to the list with active actions;
- creates a game class based on the generic *apple* class and sets the value of the *level of hunger* effect of the active action to 30.

Related scenarios: Health packs in shooters and adventure games (like *Tomb Raider*), and any edible objects in role-playing games (like *Oblivion*).

Heater

Goal: Design a heater to warm up cold hospital rooms.

Scenario: In the much anticipated genuine sequel to *Theme Hospital*, the player should again be able to place heaters inside his hospital. This time, they are not used to magically

remove mouse holes, but to keep the temperature in hospital rooms nice and steady. Game designer Jack has been given the task to design these heaters. Two restrictions: when turned on, the emitted temperature should slowly increase to a maximum temperature, and someone standing farther than 10 meters away should not feel any heat.

Events: Jack:

- assigns the *temperature* unit category (with the units *degrees Celsius* and *Fahrenheit*, among others) to the *temperature* attribute;
- defines an increase of the *temperature* attribute as an effect of the *heat* action component;
- creates a service of the act type for the *heater* class and assigns the *heat* action component as an active action;
- creates a *heater* game class that is based on the generic *heater* class;
- selects the inherited service, selects the active action *heat*;
- defines the function over time 'y=15*log(t)' to indicate that the heater slowly heats up over time t until reaching a maximum of 30 degrees Celsius (the upper limit of y in this function);
- selects a radius of 10 meters for the spatial properties, and defines the linear function over space 'y=x-3d' (where y is the resulting effect value at a particular distance d, having x as the effect value from the function of time).

Related scenarios: A bed in action-adventure games (like *Grand Theft Auto*) to slowly regain health, a shower to get wet (like in *The Sims*), or a person of which the level of hunger increases over time.

9.3 Mixture

With mixtures, demands of the substance type are thrown together, resulting in another substance as the supply. This section will give two examples.

Magic potion

Goal: Design a super effective health potion with stunning effects.

Scenario: For the fifth installment of *The Elder Scrolls* series, game designer Uriel wants to let players create their own potions at the Mage's Guild, each having a different effect when drunk. One of them is a super effective health potion that can be created by combining some very rare ingredients in the correct proportion (1:2). Unfortunately, the potion has one downside: there is a small chance that the avatar gets stunned when drinking the potion.

Events: Uriel:

- creates a *stunned* attribute and adds the states *true* and *false* to it;
- defines this attribute as an effect of the *stun* action component, changing the state to *true;*
- defines an increase of the *health* attribute as an effect of the *restore* action;
- creates two game classes of the substance type, representing the ingredients;
- creates a service of the 'mixture' type for the first ingredient;
- adds the second ingredient as a demand, and defines ratios 1 and 2 for both ingredients;

- defines the *drink* action component as an action that exhausts the target, and adds it to the list of passive actions of the first ingredient;
- assigns the *restore* action to the active actions, increasing the *health* with 100 with a chance of 0.9;
- assigns the *stun* action to the active actions, giving it a chance of 0.1;
- indicates that the chances of both actions are dependent of each other.

Related scenarios: Any potion in other role-playing or puzzle games (like *Zack and Wiki*).

Fire fighting foam

Goal: Design fire fighting foam, as fighting fires with just water is not sufficient.

Scenario: A new serious game is being created for firemen to prepare them for the real deal. Designer Sam wants to add the possibility to mix water with fire fighting foam, as this is a more effective way to fight fires than just water.

Prerequisites:
- The *extinguish* action component has been designed with several effects, like lowering the *temperature* and making the target *wet*.

Events: Sam:
- defines a new service of the 'mixture' type for the *water* class;
- adds the *fire fighting foam* as a demand;
- defines the ratio between *water* and *foam*;
- adds the *extinguish* action component as a supply;
- defines high values for the effects.

Related scenarios: Chemicals in action games (like *Alone in the Dark*) that can be mixed to create an explosive substance.

10. Conclusions and Future Work

Despite exuberant visuals, most current games considerably lack proper semantics in the objects populating their virtual worlds. Therefore, it is not possible for players to interact with virtual objects the same way as they would with objects in real life. This is partly because designing semantic objects poses especially difficult challenges, including the inherent complexity of maintaining and scaling all interactions among such objects. If game designers would have to design all semantics by themselves, and game programmers would have to implement the handling of these semantics, game development time would increase enormously. As was stated in Section 1.2, the goal of this research was to improve the semantics of objects within game worlds, by extending the behavior that is commonly found in games, resulting in more and better interaction between players and objects.

This thesis presented an approach in which *services* played a key role. As was introduced in Section 1.3, this approach consisted of five steps, which results will be summarized here:

1. Game objects have been analyzed in order to formally express them. Therefore, several components were introduced, including *classes*, *attributes*, *units*, *states*, *materials*, and *actions*. A class has been defined as a generic description of a collection of entities based on their common essential attributes. It was then possible to define an action as a process performed by an entity, yielding some attribute value changes (*active actions*) or (new) entities (*supplies*).
2. These components resulted in the formal definition of a service: the capacity of an entity to perform a particular action, possibly subject to some requirements. These requirements could be *preconditions* on attribute values, specific entities that should be available in the inventory of an entity, possibly given by another entity (*demands*), or actions that should be performed on an entity (*passive actions*).
3. To enable a game development team to incrementally specify and add services to their game objects, a three-phased methodology has been presented. In the *specification phase*, services could be specified for *generic classes*, resulting in knowledge about generic behavior. The *customization phase* was meant to define and customize *game-specific classes*, which were based on generic classes. Finally, *instances* of game classes could be created and customized in the *instantiation phase*.
4. This approach has been implemented and validated by means of a prototype, providing an integrated environment which effectively supported a simple and intuitive definition of services. In the *Library Editor*, it was possible to specify the components of the first two phases, and

establish relationships between them. Above all, services could be specified. The *Level Editor*, which was meant for the third phase, maintained an overview of a game world for which instances could be created.

5. Analogously to what a physics engine does with in-game physics, the created *Semantics Engine* was charged with all service handling during a game, taking care of requirements and actions.

The approach that was presented in this thesis has numerous advantages. By making use of generalization and inheritance, children of parent classes automatically inherit attributes and services. This drastically reduces the time to specify these classes. Furthermore, the generic classes promote reusability of previously specified object semantics. After all, generic object behavior is applicable in many games. This is extremely useful for game designers, as they are able to quickly add semantics to their game objects, without even thinking about the common object behavior for themselves. Because of this, a particular object can also assume functions or roles never anticipated by a designer. Furthermore, the presented approach easily supports behavior customization as required by each specific game, allowing the game designer to give game objects a personal touch. As everything seamlessly blends with the Semantics Engine, game programmers are not overloaded with the extra work of handling services. Instead, they are provided with a set of tools to support richer object interaction. This leads to the final advantage, aimed at gamers: with the presence of services, players are rewarded with more and better object interaction, resulting in a much deeper gameplay experience than in current games.

Enabling developers to create game objects that are aware of each other's services will be instrumental to achieve more and better object interaction, and to improve the gameplay. However, it should also be stressed that object semantics isn't but a (powerful) means to serve the gameplay. In particular, it will never automatically make dispensable the creative work of designers. On the contrary, care should be taken to avoid overloading objects with superfluous semantics, as semantics make virtual objects not only more realistic, but more complex as well, which could end up undermining the gameplay. Therefore, it is the task of the game designer to seek a balance between achieving realism and good gameplay.

As far as the goal of this MSc research is concerned, an attempt was made to improve the semantics of game objects. The approach presented here, giving designers the possibility to include realistic semantics by means of services, while keeping much control on the fine-tuning of the behavior of their objects, is a valuable aid in that direction. Although it was unfortunately not possible to test the approach in a real game, the Tech Demo successfully demonstrated its strong potential. Hopefully, Tim Tutenel will put all work to good use, as this research was part of his PhD work. In his work, Tim strives for the creation of adaptive virtual environments with objects that have dynamic behavior, in order to create a more immersive interaction experience. He tries to accomplish this with semantic information, for which services are very useful. When this research is finished, parts of the prototype (mainly the Library Editor and the Semantics Engine) will be integrated into the implementation of his work, where it can be used and improved when necessary.

During the research, it was not possible to take every aspect of object interaction and services into consideration. Especially the area of substances and mixtures is open to improvement. Defining a quantity for substances in the instantiation phase is possible, but unlike physical objects, they cannot be represented by a 3D model, which makes their visual representation, in combination with semantics, much more difficult. Furthermore, a mixture between water and sand, for example, will not instantly result in mud. Instead, both substances should gradually mix. For both problems, advanced fluid simulations are required. Semantics of substances could then indicate whether a substance mixes with another substance (resulting in a suspension) or not (an emulsion), and what the resulting mixture is, in case of a suspension. Transitions between physical states is another useful addition, to define a substance in gas, fluid, and solid form (e.g. vapor, water, and ice).

The aggregation ontology defined physical relations between parts and the assembly they are in. However, it is also possible to define relations between assemblies (or unrelated entities without parts), which can be called *connections*. Electrical devices, for example, will demand electricity. As electricity cannot just be given (handed over) to these devices, a power cable is required that connects an electrical device to some power source, and transmits the supplied electricity from one side to the other. A sewer system is a similar example. When this notion is added to the current design, an entity should become aware of the entities to which it can be connected, and on which position (on the 3D model) this connection should be made. A possible implementation in a game might be a puzzle in which the player should create connections between objects in order to solve the puzzle. Connections can also be used for services. For example, in order to get a soda from a vending machine, the machine should not just be powered on, but connected with a power cable to a socket as well, which can be done by the player if necessary.

The prototype is not flawless either. Some improvements will be listed here. First, the plain text files that make up the current database could be replaced by more advanced database files, as this can make storage and retrieval easier and faster. Second, for parts and wholes, the Library Editor could be extended with the functionality to indicate on a 3D model of a whole where its parts should be situated. Something similar can be done for the inventory of a game class (and when the design is extended with connections, for them as well). A third improvement is meant for the Level Editor, which could be slightly adjusted in such a way, that it can be integrated into several commonly used level editors. Another improvement is to achieve more usability and flexibility in the editors. A fifth improvement is the implementation of faster algorithms in the Semantics Engine, in order to give less overhead in a game. Furthermore, much more research could be done towards a collaborative approach to populate the generic libraries, as was discussed in Section 3.3. Finally, the engine could be experimentally coupled to a game AI system, to show that artificial agents can make use of services as well, in addition to players' avatars.

In the future, the above mentioned additions and improvements to the design and prototype could be carried out in further research. This way, services can be made even more useful than they are now. Anyway, this shows that the services that were presented in this thesis provide a solid foundation for achieving more and better object interaction. Hopefully, this will open up the next chapter in next-gen gaming.

Bibliography

Barnes, D. J. (2000). *Object-Oriented Programming with Java: An Introduction.* New Jersey: Prentice Hall, Inc.

Bethesda Game Studios. (2006). *The Elder Scrolls IV: Oblivion* . Bethesda Softworks. Platform: PC/Xbox 360.

Bilas, S. (2002). Data-Driven Game Object System. *Game Developers Conference.* San Jose, USA.

BioWare. (2002). *Neverwinter Nights* . Infogrames/Atari. Platform: PC.

Bureau International des Poids et Mesures. (2008, October 21). *BIPM - SI*. Retrieved July 17, 2009, from The International System of Units (SI): http://www.bipm.org/en/si

Capcom. (2007). *Zack & Wiki: Quest for Barbaros' Treasure* . Capcom. Platform: Wii.

Church, D. (2007, October 27). *Object Systems: Methods for Attaching Data to Objects.* Retrieved September 16, 2008, from Game Object Systems - Chris Hecker's Website: http://chrishecker.com/Game_Object_Systems

Cyan Worlds. (1994). *Myst* . Brøderbund. Platform: PC.

Duran, A. (2003). Building Object Systems: Features, Tradeoffs, and Pitfalls. *Game Developers Conference.* San Jose, USA.

Eden Studios. (2008). *Alone in the Dark* . Atari. Platform: PC/PS2/PS3/Wii/Xbox 360.

Gösele, M., & Stuerzlinger, W. (1999). Semantic Constraints for Scene Manipulation. *Proceedings Spring Conference in Computer Graphics*, (pp. 140-146). Budmerice, Slovakia.

Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition, 5* (2), 1999-220.

Havok. (2009, July 17). Retrieved July 17, 2009, from Havok: http://www.havok.com/

Huhns, M. N., & Singh, M. P. (1997). Ontologies for Agents. *IEEE Internet Computing , 1* (6), 81-83.

Infocom. (1980). *Zork* . Personal Software/Infocom. Platform: Amiga/Atari/Commodore/PC.

Jeff Tunnell Productions. (1992). *The Incredible Machine* . Dynamix. Platform: PC.

Kallman, M. (2001). *Object Interaction in Real-time Virtual Environments.* PhD Thesis, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland.

Kallmann, M., & Thalmann, D. (1998). Modeling Objects for Interaction Tasks. *Proceedings of the 9th Eurographics Workshop on Animation and Simulation (EGCAS)*, (pp. 73-86). Lisbon, Portugal.

Maxis. (2000). *The Sims* . Electronic Arts. Platform: PC.

Microsoft (a). (2009, July 17). *Microsoft Visual Studio 2008*. Retrieved July 17, 2009, from Team System - Application Development: http://www.microsoft.com/visualstudio

Microsoft (b). (2009, July 17). *XNA Creators Club Online*. Retrieved July 17, 2009, from Home: http://creators.xna.com

Microsoft (c). (2009, July 17). *Xbox.com Home*. Retrieved July 17, 2009, from Xbox: http://www.xbox.com

Miller, G. (1995). WordNet: A Lexical Database for English. *Communications of the ACM , 38* (11), 39-41.

Nintendo. (2009, July 16). *Wii at Nintendo*. Retrieved July 17, 2009, from Wii: http://www.nintendo.com/wii

People Trading Services. (2008, August 1). Retrieved July 19, 2009, from Barter Your Services and Trade Services: http://www.peopletradingservices.com

Peters, C., Dobbyn, S., MacNamee, B., & O'Sullivan, C. (2003). Smart Objects for Attentive Agents. *Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG2003).* Plzen, Czech Republic: Science Press.

Smith, G., Salzman, T., & Stuerzlinger, W. (2001, June). 3D Scene Manipulation with 2D Devices and Constraints. *Graphics Interface* .

Taito Corporation. (1978). *Space Invaders* . Midway. Platform: Arcade.

Tutenel, T., Bidarra, R., Smelik, R. M., & de Kraker, K. J. (2008). The Role of Semantics in Games and Simulations. *Computers in Entertainment , 6* (4).

Tutenel, T., Smelik, R. M., Bidarra, R., & de Kraker, K. J. (2009). Using semantics to improve the design of game worlds. *Proceedings of the 5th Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2009).* Stanford, USA.

Wikipedia. (2009, July 22). *Game engine*. Retrieved July 27, 2009, from Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Game_engine

Wikipedia. (2009, July 14). *Supply and demand*. Retrieved July 17, 2009, from Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Supply_and_demand

WordNet. (2009, February 16). *WordNet documentation*. Retrieved July 17, 2009, from WordNet: http://wordnet.princeton.edu/wordnet/documentation/

Xu, K., Stewart, J., & Fiume, E. (2002). Constraint-Based Automatic Placement for Scene Composition. *Proceedings of Graphics Interface '02* , 25-34.

Appendix A. Projects and Classes Overview

The prototype has been implemented using C#, WPF, and XNA. With Microsoft Visual Studio 2008, one solution was created, consisting of several projects. This appendix will describe the purpose of each project, and give a high-level description of the most important classes, user controls, and related files.

SemanticData

This project contains all semantic data, including the C# classes that are used to store the components that were described in Chapter 4. Furthermore, it contains C# classes for everything that is service-related, as was discussed in Chapter 5. In addition to several tools, the project also contains the database of Section 6.5.3, and the Semantics Engine of Chapter 7.

Action	An action component which keeps track of its effects, and whether it exhausts the actor or target.
ActiveAction	One of the possible service actions. Stores information about an action that can be performed, including temporal and spatial properties.
Attribute	An attribute component, having knowledge about the units or states in which its value can be expressed.
Class	A generic class component, containing materials, attributes, parts, wholes, and services (and units in case of a substance).
ClassCategory	A class category contains classes that meet certain conditions. It is possible to add services to all nodes in a category.
Demand	A requirement that is based on a class component. Also contains the required quantity of the demand, and its preconditions.
Effect	An effect of an action component, based on an attribute. Stores how the attribute value is changed.
GameObject	The implemented name of a game-specific class component. When based on a generic class, it will inherit its parts, wholes, materials, attributes, and services. These can be defined separately as well. It has an inventory and also stores the names of its models, animations, and sounds.
Instance	An instance component, that is created from a game-specific class. It therefore inherits all attributes and services, but it also has its own position, rotation, scale factor, and alpha value, among others.
Material	The material component indicates the matter of which an entity consists, and therefore contains material-related attributes.
Node	A node is a super node with parents and children.
Part	A class or instance that is part of a whole.
PassiveAction	A requirement of a service, which stores an action that should be performed on the entity with the service, the animation that should be played by the entity when performed, and the preconditions on the actor.

Precondition	A precondition is an attribute that should have a particular value, defined by the inequality sign.
Requirement	An abstract class for the requirement of a service. Stores a serial number for the sequence of interaction steps.
Service	A service that has preconditions, demands, and passive actions as requirements, and supplies and active actions as service actions.
ServiceAction	An abstract service action, storing the chance that this action can be performed, and a serial number for the sequence of interaction steps
State	A state component, storing the states to which it is related.
SuperNode	A super node defines a component and stores its names and description.
Supply	A supply that is given by a service in a particular quantity. Also defines whether it should be newly created, or given from an inventory.
Unit	The unit component is aware of its category, and has the equations to convert to and from the base unit of its category.
UnitCategory	A unit category can have multiple units and a base unit.
Whole	A class or instance that is a whole.

Chance	A chance with a value that can be dependent on an attribute and other chances.
Duration	Stores the temporal property of a service action. Also includes a value and unit for a fixed time period.
Enums	A collection of many enums that are used to define all possible warning messages, class types, service types, supply sources, demand usages, etc.
Equation	Stores a string representation of an equation. Makes use of the MathParser to convert a variable to an outcome.
MathParser	A set of classes that can be used to parse an equation. When given an equation and a variable, it will calculate the outcome.
ModelInfo	Stores the name of a model and its preconditions.
Range	Stores the spatial property of a service action. Also includes a value and unit for a radius.
Value	A value with an optional prefix and unit.

Database	The database is able to load, modify, and save (the relations between) classes, actions, attributes, units, states, materials, game objects, and instances. For each component, the database makes use of a data and relations file.

SemanticAction	An insulated version of an active action, which should be used by game programmers when they access the Semantics Engine.
SemanticInstance	An insulated version of an instance, which should be used by game programmers when they access the Semantics Engine.
SemanticsEngine	The Semantics Engine keeps track of all instances in the game, and takes care of all service handling. Furthermore, it contains methods to make instances perform regular and special actions, and smart helper methods that can be used for artificial agents.

LibraryEditor

The C# and XAML source code of the Library Editor can be found in this project. Section 6.2 described the editor in much detail.

LibraryEditorWindow	A resizable window that contains the user control of the Library Editor.
LibraryEditorUserControl	In the Library Editor, library components can be created, modified, and removed. For example, materials can be related to attributes, which can in turn be related to units and states. For classes, services can be defined, including their requirements and actions. Furthermore, it is possible to define game-specific classes, and base them on generic classes.

LevelEditor

This project contains all the files for the Level Editor that was discussed in Section 6.3.

FlyingCamera	A camera through which the game world can be observed from a first person view. The WASD keys are used to move around, while a mouse movement results in a change of orientation. With this camera, it is possible to fly through the entire game world, ignoring collisions and physics.
LevelEditorWindow	A resizable window that contains the user control of the Level Editor.
LevelEditorUserControl	In the Level Editor, it is possible to create instances from game objects. Attributes and services of instances can be customized even more, if desired. Furthermore, instances can be given a position and rotation in the world, and a scale factor and alpha value can be defined.
ILevelEditor	An interface for the game screen of the Level Editor. It defines the methods that should be implemented when creating this screen. Having such an interface allows game developers to integrate their game engine and visualization with the service-oriented user control of the Level Editor.
LevelEditorXNA	This is a screen in the Level Editor that shows a game world that is created with the XNA framework.

SharedEditorResources

The Library Editor and Level Editor have many similarities, both in functionality and visual style. Therefore, this project contains all the resources that are shared by both editors. The Flow Diagram and Graph Plotter were discussed and shown in Section 6.4. In addition to the files listed below, this project also contains numerous icons that are used for all the buttons in both editors.

AboutBox	A popup box that shows general information about the application.
FlowDiagram	In this diagram, requirements and actions of a service can be ordered, resulting in a sequence of interaction steps.
GraphPlotter	A window in which predefined equations can be selected and modified, after which they are shown as a graph. Creating custom equations is possible as well.
Labels	A resource file that contains all the text labels in the GUIs. If desired, the English labels can easily be replaced by labels in a different language.

Messages	A resource file that contains all English warning messages when a particular operation fails.
ResourceDictionary	This dictionary contains many data templates for the listbox items in the editors. Furthermore, it defines all background colors.
ToolTips	A resource file with all the English tool tips that are shown when the mouse cursor is hovered over a GUI item.

TechDemo

This project is the implementation of the Tech Demo that is described in Chapter 8. Besides a game class (*ServiceGame*), it contains helper classes for the player, and support for object selection and interaction.

CircularList	A circular list in which it is possible to move to the previous or next entry. When at the last entry, the next one will be the first, and vice versa.
FirstPersonCamera	A camera through which the game world can be observed from a first person view. The WASD keys are used to move around, while a mouse movement results in a change of orientation. Movement in the Y-direction is not possible.
ServiceGame	This game is a tech demo to show how a player can interact with objects for which services have been defined. It contains a simple 3D world in which several objects are placed. By using the mouse, actions can be performed.

SharedGameResources

As the virtual world of the Tech Demo has been created with the Level Editor, both projects share several resources. This project contains these resources, including the textures for the skybox and the terrain, and the 3D models that represent the game instances.

Camera	A camera with basic functionality, like translation and rotation.
GameInstance	A game instance is used to store both semantics and physics information. It also contains a 3D model that can be drawn.
SkyBox	The skybox is a cube with a background image that creates the illusion of distant surroundings and the sky.
Terrain	A 3D terrain that is created from a 2D height map.

ModelViewer

The viewer in this project can be requested in the Library Editor when assigning a 3D model to a game-specific class. As the name implies, the viewer shows the model, rotating around the Y-axis. The Model Viewer only supports models in FBX and X format. If the prototype would be extended with the functionality to define how parts are positioned and oriented with respect to their whole, this viewer could serve as a base.

HeightmapProcessor

This project contains several files to generate a textured 3D terrain from 2D height map and texture files. This terrain is used in the Level Editor and the Tech Demo.

Appendix B. Paper

Title
Services in Game Worlds: A Semantic Approach to Improve Object Interaction

Authors
Jassin Kessing, Tim Tutenel, Rafael Bidarra

Keywords
Game worlds, services, semantics, object interaction

Topics
Game design, video games

Conference
The 8th International Conference on Entertainment Computing, ICEC 2009

Date
3-5 September 2009

Location
Conservatoire National des Arts et Métiers (CNAM), Paris, France

Website
http://icec2009.cnam.fr/

Category
Short technical paper

Editors
S. Natkin, J. Dupire

Publisher
Springer

Proceedings
ICEC 5709, pp. 276-281, 2009

Services in Game Worlds: A Semantic Approach to Improve Object Interaction

Jassin Kessing, Tim Tutenel, Rafael Bidarra

Computer Graphics Group
Delft University of Technology, The Netherlands
JassinKessing@gmail.com, T.Tutenel@tudelft.nl, R.Bidarra@ewi.tudelft.nl

Abstract. To increase a player's immersion in the game world, its objects should behave as one would reasonably expect. For this, it is now becoming increasingly clear that what game objects really miss is richer semantics, not eye-catching visuals. Current games' lack of semantics is mostly due to the difficulty of game designers to realize such complex objects. This paper proposes a solution to this problem in the form of services, characterizing classes of game objects. An example of this is the service of a vending machine, which exchanges a coin for a soda. A three-phased methodology is presented to incrementally specify and add services to game objects. This approach has been implemented and validated by means of a prototype system, which enables a simple and intuitive definition of services in an integrated environment. It is concluded that game objects aware of their services facilitate more and better object interaction, therefore improving gameplay as well.

Keywords: services game worlds, services, semantics, object interaction

1 Introduction

Look around and you will probably see objects scattered all around the place. If the same room would be used as the virtual environment of a game, one would probably want to see the same objects - or at least some objects - because empty environments are unnatural to walk through. Game environments that are filled with objects will therefore help immerse the player into the game world. By using graphics, animations, and physics, virtual objects could appear as players expect. However, that only accounts for their visual aspect, because most objects in games are still useless, being there for decoration purposes only. Only few objects, which are crucial for the game progress, are made functional.

An example is the role-playing game *The Elder Scrolls IV: Oblivion* [1], where objects can be picked up and used on specific locations to trigger an event, like opening a door with a key. Some objects have an effect on the player's avatar, for instance eating bread to increase the health, or wearing armor to increase the level of defense. In the game *Alone in the Dark* [2], several objects, which look useless at first sight, can be combined to create a functional object: a full battery and an empty flashlight will provide a light in the dark when combined.

In the examples above, the functionality of each object (including its meaning, roles, etc.) was thought up by the game designer and implemented by the programmer. In the real world, a particular object may assume other functions or roles never anticipated by its designer; with a game object in a virtual world, this is definitely not (yet) the case, and certainly not automatically. This limitation makes it impossible for a player to (make his avatar) interact with a game object in many reasonable ways.

In the fields of linguistics, computer science and psychology, *semantics* is the study of meaning in communication. When focusing on virtual environments for computer games, semantics is the information conveying the meaning of (an object in) a virtual world [3]. A serious problem in current game development is a lack of tools to easily specify and add semantics to objects, resulting in a lack of object semantics in games. With a semantically rich object representation, virtual objects assume behaviors like in the real world, instead of consisting of a geometric model only. This can be illustrated with a few examples. When eaten by a character, an apple will reduce the hunger level of that character; in other words, an apple provides the *service* of satisfying someone's hunger. A coat serves its wearer for warmth. A fire, however, will provide warmth to everyone in the area. And a vending machine has the service to supply cans of soda, but only after it has received money.

The role of semantics in virtual environments is receiving increasing attention [3], but so far not much research has been done on adding semantics to game objects, let alone with the purpose of making them more functional or improving the overall gameplay. This paper focuses on our research efforts to improve the semantics of objects placed within game worlds. In particular, the notion of services is proposed, by which virtual objects get to 'know' about their roles in the world, how they can affect other entities (including the player's avatar or artificial agents), and how others can interact with them. Empowered with the notion of services, objects acquire their own behavior, instead of a purely predefined behavior; they can behave as one expects, and correspondingly one is able to interact with them as one expects, regardless of whether the virtual object is completely imaginary or mimicking some real world object. We believe that this can significantly change and improve the gameplay, as players will be able to express their creativity and find more paths to achieve the same goal. Enabling game developer teams to easily declare services and assign them to object classes, as described in this paper, is a major step towards the ultimate goal of achieving more and better object interaction.

2 Related Work

Smart objects [4] were a successful proposal for adding semantics to virtual objects, dealing with many of the possible user interactions in a virtual environment. Noticeably, smart objects were primarily devised for manipulation, animation, and planning purposes, like grasping, pulling, or rotating (individual parts of) objects. An example is an artificial agent that can open a door by moving its hand to the door knob, using the correct hand posture, and turning the knob. Although smart objects are powerful for these purposes, they lack the information of which services they provide to their users.

Research in artificial intelligence (AI) proposed the notion of ontologies, due to the lack of shareable and reusable knowledge bases. An ontology is an explicit specification of a conceptualization: a representational vocabulary for a shared domain of knowledge, in the form of human-readable and machine-enforceable definitions of classes, relations, functions, and other objects [5]. When placing ontologies in the context of this research, they define the meaning of objects and the relations between them. In ontologies, important relationships are *generalization and inheritance*, where classes are connected, and each subclass inherits the features of its superclass [6]. The class *Car*, for example, has the class *Vehicle* as its parent. Another important relation is *instantiation*, which relates a class with each of the individuals that constitute it. A *Ferrari*, for instance, is a kind of *Car*. In Section 4, the usefulness of these two relations for the creation of objects with services will become apparent.

3 Designing Services

In order to design services for game objects, it can be very useful to analyze how real world objects can be structured and classified. In particular, we can identify the notions of *class* and *attribute*. Each object in the real world can be said to belong to some class, defined as 'a generic description of a collection of entities based on their essential common attributes'. An attribute, in turn, is defined as 'a characteristic of an entity'. Classes, therefore, describe entities, varying from physical objects like 'apples' and 'people', to substances like 'water'. For attributes, one can think of abstract attributes like 'edibility', or physical attributes like 'mass'. Units and states express the values of attributes. The 'mass' attribute could be expressed in the units 'kilograms' or 'ounces', while 'edibility' could be expressed in the states 'edible' or 'inedible'. Units are also required to express substances, because they are not quantifiable in integer values only, unlike physical objects.

The notions introduced above give us a foundation for the definition of services. In the real world, entities have particular functions, and provide services, and this should also be the case for entities in a virtual world; for example, a coat provides the service of supplying warmth, but only when it is worn. We define a *service* as 'the capacity of an entity to perform an action, possibly subject to some requirements'.

An *action* can then be described as 'a process performed by an entity, yielding some attribute value changes or (new) entities'. Actions are best illustrated by some examples. A service of a heater is to heat the entities in the surrounding area. This means that the values of their temperature attribute rise; see Fig. 1. Attribute value changes do not have to occur to target entities only; they can also affect the actor. Consider an avatar punching an enemy, which lowers the enemy's health, but also increases the avatar's fatigue. The service of a vending machine, supplying a soda, is a good example of an action yielding an entity; see again Fig. 1. This soda is an entity that is supplied from the inventory of the vending machine. However, it is not necessary that the actor always has a stock of existing entities that can be supplied, as an entity's action can also yield new entities. An example is a saw machine that requires trunks, and processes them into wooden planks, which are new entities. This process leads us to the notion of *service requirements*: they can be either actions (e.g. the coat should be worn, and the saw machine should be

given trunks) or some attribute constraints, as for example a range of values/states (e.g. electrical devices should be powered on before performing an action, and the fatigue level of the avatar should not be too high before being able to fight).

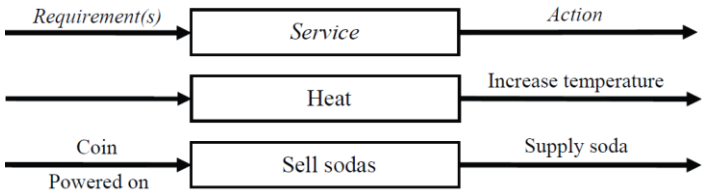

Fig. 1. A generic service, a service of a heater, and a service of a vending machine

From the examples above, there are four important elements that should also be taken into account when designing services. First, quantities are essential to indicate how many entities are exchanged during an action, or in which amount an entity is exchanged, in case of substances. Second, temporal properties are relevant, because they indicate the duration of a service, which could be a one-time event, or last for some amount of time. Third, spatial properties indicate who or what is affected by a service, e.g. the consumer, or all entities within a certain radius. Finally, a sequence of interaction steps indicates the order in which requirements should be met before performing an action.

4 Services Put to Work: A Three-phased Approach

The concepts developed in the previous sections have been implemented in a prototype system which supports the definition of services for game objects step by step. This system covers the three main phases that were identified in the object design process: (i) a *specification phase*, in which generic classes are specified in a library, (ii) a *customization phase*, where a selection of classes from that library is customized into concrete game-specific classes, and (iii) an *instantiation phase*, where object instances of these game-specific classes are placed in a game world. Figure 2 gives an overview of these three phases.

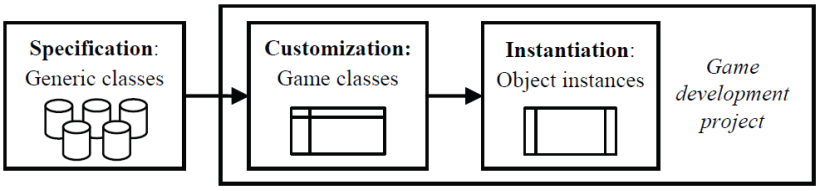

Fig. 2. A phased approach with generic classes in the specification phase, game-specific classes in the customization phase, and object instances in the instantiation phase.

The key idea of the specification phase is to create a library of generic semantic classes. By designing generic classes that can be used in all kinds of virtual worlds, consistency and reusability are stimulated, and thus development time reduced. In this phase, libraries of classes, attributes, units, states, and actions are created. Relations can be established among these components, and for each class, services can be defined in order to specify its semantics. By applying inheritance, a class hierarchy is developed, with attributes and services that have been assigned to a class being inherited by all its children. In this way, for example, an attribute like 'mass' does not have to be defined for each single class, but to the 'physical object' class only. To populate these libraries, the WordNet database [7] was used, as it contains many nouns and verbs in the English language, being therefore useful for many possible classes, attributes, actions, etc.

In contrast with the specification phase, the other two phases are not generic. Instead, customization and instantiation play a central role during each particular game project, which typically has its own unique environment, object style and desired behavior, etc. In the customization phase, specific game classes are derived from the generic classes from the first phase, thus automatically inheriting all their generic semantics, including their set of attributes, and also the services they provide. It suffices then to customize the specific behavior desired for this particular project, including their specific attribute values, e.g. quantities and temporal properties. The customization phase is also the right time to assign the project-specific 3D models to the relevant game classes. Finally, in the instantiation phase, instances of the customized game classes can be created and placed inside a game world, which is done by means of a level editor specifically created for this purpose.

5 Conclusions

Despite exuberant visuals, most current games considerably lack proper semantics in the objects populating their virtual worlds. This is partly because designing semantic objects poses especially difficult challenges, including the inherent complexity of maintaining and scaling all interactions among such objects. This paper presented a solution to that problem in the form of services, specified as characteristics of classes of objects. A three-phased methodology has been presented that enables a game development team to incrementally specify and add services to game objects. This approach has been implemented and validated by means of a prototype system, providing an integrated environment which effectively supports a simple and intuitive definition of services. Among the numerous advantages of this approach, among them (i) it promotes reusability of previously specified object semantics, (ii) it easily supports behavior customization as required by each specific game, and (iii) it seamlessly blends with our semantics engine, charged with all service handling during the game (analogously to what a physics engine does with in-game physics).

We believe that enabling designers to create game objects that are aware of each other's services will be instrumental to achieve more and better object interaction. This in turn is considered one of the key conditions to significantly improve gameplay. However, it should also be stressed that object semantics isn't but a (powerful) means to serve the

gameplay. In particular, it will never automatically make dispensable the creative work of designers. On the contrary, care should be taken to avoid overloading objects with superfluous semantics, as semantics make virtual objects not only more realistic, but more complex as well, which could end up undermining the gameplay. Therefore, it is the task of the game designer to seek a balance between achieving realism and good gameplay. The approach presented here, giving designers the possibility to include realistic semantics by means of services, while keeping much control on the fine-tuning of the behavior of their objects, is a valuable aid in that direction.

In the future, we would like to experiment with coupling our semantics engine with a game AI system, so that artificial agents can make use of services as well, in addition to players' avatars.

Acknowledgement. This research has been supported by the GATE project, funded by the Netherlands Organization for Scientific Research (NWO) and the Netherlands ICT Research and Innovation Authority (ICT Regie).

References

1. Bethesda Game Studios: The Elder Scrolls IV: Oblivion. Bethesda Softworks (2006)
2. Eden Studios: Alone in the Dark. Atari (2008)
3. Tutenel, T., Bidarra, R., Smelik, R. M., de Kraker, K. J.: The Role of Semantics in Games and Simulations. In: Computers in Entertainment, vol. 6 (4). ACM, New York (2008)
4. Kallmann, M., Thalmann, D.: Modeling Objects for Interaction Tasks. In: Proceedings of the 9th Eurographics Workshop on Animation and Simulation, pp. 73-86. Lisbon (1998)
5. Gruber, T. R.: A Translation Approach to Portable Ontology Specifications. In: Knowledge Acquisition, vol. 5 (2), pp. 199{220. Academic Press Ltd., London (1993)
6. Huhns, M. N., Singh, M. P.: Ontologies for Agents. In: IEEE Internet Computing, vol. 1 (6), pp. 81{83. IEEE, Piscataway (1997)
7. Miller, G.: WordNet: A Lexical Database for English. In: Communications of the ACM, vol. 38 (11), pp. 39{41. ACM, New York (1995)