

# Welkup Presents: Designing Super Smash Bros Classic (v3.0)

---

An Introduction to Style in GML

*Matt Soukup*

## *Introduction*

When I first got into programming games, I didn't even know how to program. Games were, essentially, my segue-way into the programming universe. When I stumbled upon Game Maker circa 2000, I couldn't believe my good fortune. Seven years and four versions later, I'm still using Game Maker, and I've never looked back. I've since been exposed to many other object oriented programming languages and have begun to notice Game Maker's trite peculiarities. Instead of scoffing at Game Maker's seemingly inherent lack of functionality in some important areas, I embrace what Game Maker is and use creativity to coax that same functionality into existence. To me, programming is an art form<sup>1</sup>, and Game Maker is just one of many creative mediums.

Even though Game Maker does not provide all of the functionality of a normal programming language, steps can be taken to reduce the disparities. The goal of this document is to provide a style of GML programming that truly helps reduce the complexity of a project by taking those steps. This style of programming is most helpful in managing larger projects. The key is in the organization of the Game Maker file -- the style.

Super Smash Bros Classic will be the teaching tool throughout this paper. It has a long history of development and has the potential to be a great teaching tool. As long as I've been programming games, I've been programming an NES-rendition of Super Smash Bros. Throughout the years, it has gone through name changes, version changes, and countless other transformations. It has a history of sloppy code and an adherence to, at best, terrible design, both characteristics of an amateur program. Not to worry, though, as this project has now happily adapted itself into a case study of sorts, aiming to teach the reader what not to do while providing many useful design strategies along the way.

## *Principles of Software Design*

### **Design Concepts**

A major goal in design is encapsulation. While the word means different things to different people, encapsulation basically creates separate entities whose underlying implementation is largely hidden. For example, encapsulation applied to a car class means that car class is going to contain means for braking, accelerating, turning on wipers, etc., and you will only know that it works, not how it's implemented. They are separate in that they should depend on as few other classes as possible. While it is alright to use very low-level classes frequently in your programs, a concept called high fan-in, having one class interact with too many other classes, high fan-out, should be avoided. Classes should have one major direction to encourage cohesion. Cohesion is the strength of a class' core competency. It ensures that a class does not attempt to do too many different things.

The ultimate purpose of defining all of these terms is to reduce complexity. Programs grow to unfathomable sizes, meaning its going to be difficult for the programmer to be able to "take it all in." That is why every step of the process should be asking, "What can I do to make this program less complex?"

### **Development Models**

The industry has a number of common approaches to software design called development models. I will cover three common methods: the waterfall method, the prototype method, and the iterative method. This is just a short list, and many others exist.

The waterfall method is the most famous approach and attacks the problem linearly. It consists of a number of concrete, sequential steps. Finishing one of these steps is like crossing the Rubicon, there's no going back. It's a simplistic approach but is certainly better than no approach at all. The steps are analyzing requirements, designing, coding, testing, and maintaining. The design step tends to take the longest since it has to be specific enough to build the foundation that you will be coding on.

The prototype method builds a working draft of the final product; then, after receiving feedback from the client, it throws the draft away and starts anew. This is the method being applied to Smash Bros Classic. Demo v2.01 is the prototype. It provided us with a working example, and from the player feedback, we learned. Now we are rewriting the game with the big picture in mind. A major piece of the prototype method is the need to throw away the original code. Sometimes companies would rather not do this. To prevent the temptation of

using the prototype code for the final version, the prototype is often written in an entirely different language. We are still using Game Maker for the final version, however, because of its familiarity. Creating games with Game Maker can be a very swift process. This makes Game Maker an excellent medium for creating prototypes.

The iterative method is the most common approach to the design process. It builds upon itself over a number of iterations. This design type is most commonly used because it represents flexibility. The idea is to have a certain number of features completed in a given iteration. This iteration will be functional on its own. The next iteration will build off the previous and add more features. If the new additions work, keep that iteration. If they don't, roll it back. In this way, the new iteration should be mostly distinguishable from the previous.

### *Foundations*

This section will cover some basic stylistic practices that can be used with Game Maker. It provides the groundwork for some of the more advanced ideas.

#### **Resource Naming**

While we have all been told to name our objects and sprites in a logical fashion (such as by adding "spr" as a prefix to sprites), resource naming carries much more weight than readability. For readability's sake, in Smash Bros. Classic v2.01, we used the prefix "spr" for sprites and "l" or "r" for direction suffixes. Objects were headed with "obj" and rooms with "room." The other resources did not follow much of a pattern, but this will be set straight in v3.0. A way to take your naming conventions to the next level is presented in the following example.

Consider the script from v2.01 called `check_facing()`. This script takes in a string such as 'walk' and changes the character's sprite to the walking sprite. Notice how it doesn't care which character is using it. The key? Two things: careful sprite naming and `execute_string()`. Here is the gist of the `check_facing` script:

```
var direct;
if (face_right == true)
{
    direct = "r";
}
else
{
    direct = "l";
}
```

```
}  
  
execute_string("sprite_index = spr" + string(name) + string(argument0)  
+ direct + ";" );
```

An implementation like this allows maximal reuse of code, which is always a good thing. When you include a lot of low level functions like `check_facing`, you can create a character father class. Once you've established your bases inside this father class, new characters can inherit from it. This is just one step towards making programming in new characters a snap!

### Variable Use

You might have heard that the road to bad programming is paved with global variables, but I'm here to tell you that this old adage does not apply to Game Maker. Let's look at a few reasons that they are bad in other languages. The most common argument: Global variables have no locality, anything can edit them, which creates huge dependencies throughout the program.<sup>2</sup> For this, I can only say that they should be used as sparingly as possible, and there should be extensive documentation about each one's use, such as where it is initialized and what objects use it. Second argument: They increase complexity because they can exist anywhere throughout the program. Our solution to the first problem helps this a bit. Game Maker uses the prefix **global** for its variables, so it makes it obvious that they are, in fact, global.

Despite the cautions, I argue that they are essential to Game Maker. Since Game Maker is room-based, it is the only way to keep track of variables between rooms, minus something slow like file I/O. This is a critical issue and can only be remedied by using a single room. This is a step backwards, though, as multiple rooms helps reduce complexity. Let's do ourselves a huge favor and limit global variables as much as possible.

While we want to limit global variable use, we should always use local variables when possible! By local variables, I mean variables declared with the **var** keyword. This stores them in the registers instead of memory, which increases their speed.

### Game Maker Data Types

Game Maker is not a very strongly typed language. This means that, unlike in C, we do not need to declare each and every variable with a preceding type. Game Maker will recognize what we are doing and allocate enough space on its own. (This is similar to the Python

programming language.) With that said, the programmer should still make an effort to use a fixed set of different types in order to reduce complexity.

According to the Game Maker manual, Game Maker recognized two types, strings and real values. This is inadequate for me. Even if Game Maker represents all of these in the same way, it would still be helpful to differentiate. The following is a table of the different types of values seen in Game Maker:

Name	Denoter	Comment
Integer	int	Whole number
Floating point number	float	Number with decimal
Boolean	bool	True or False (Nonzero or zero)
Instance identifier	instance_id	ID of an object instance
Resource identifier	resource_id	ID of a resource (sprite, object, etc.)
String	string	String of characters (use “ ”) <sup>3</sup>
Character	char	Single character (use ‘ ’)

This table can be useful for defining an objects implementation in its Create event. (See the Constructors section of the Class chapter.)

### Modularity of Files

Thanks to Game Maker’s file merging capabilities, we can take modularity to the next level. Let me present you with a scenario that I have encountered many times. You are part of a game design partnership. One of you codes while the other sprites. One day, on a whim, you two get together and start working on a great new game. Each does his part at the same workstation while the other sits observing. (This concept is called extreme programming when both people are programming in this way.) Eventually, the game becomes more and more complicated, and working together becomes less and less practical. You start working separately, but your files are never really in sync, and merging them at the end of the period would be tedious and required a lot of reworking. Had you a chance to redo it, you might try to modularize your files better; that is, separate them into entities that will be merged upon completion.

Each separate file, when merged with a father file, will create a folder in each of its resource areas. The folder will have the same name as its editable file parent, which makes identification much easier on the designer. (See the “Tools” section for notes on a related aspect of synchronization called SVN.) This may not always be a practical solution, but it should be used whenever possible.

## Class Design

While classes are not as formally defined in Game Maker, users can get most of a class' benefits through the use of objects. An object in Game Maker, like a class in C++, is a template that defines the characteristics associated with whatever the object is representing.

Classes in other languages are able to house methods, functions specific to that class. Scripts are Game Maker's analogy to functions, but they are not localized to a particular object. This can be bypassed somewhat by keeping a strict folder system in the Scripts resource area. Each object could have a folder in the library of scripts devoted to it. So now my Kirby object can have all of its scripts together in a folder called "objkirby." This is somewhat analogous to packages in Java.

### Constructors

Normally, classes have methods known as constructors that allow the programmer to create an instance of the object. The corresponding "constructor" in GML is

```
instance_create(x, y, object_name);
```

Then, along with the Create event, the object is initialized. Constructors typically take in arguments that will initialize the object's data members (variables). These tools give no elegant way of doing this straight up. What we need is a formal way of defining a constructor using a script.

Let's say I wanted to create a new script to represent a constructor for an objlinkarrow. Link's arrows have a few characteristics that need to be established when they are created. First, x and y positions have to be established. Then, it needs to know whether it is firing right or left. Finally, it needs to know who created it. I could name it constr\_objlinkarrow() and define it as such:

```
with (instance_create(argument0, argument1, objlinkarrow))
{
    creator = argument2;
    direction = argument3;
    flinch_direction = 180 - 180 * direction;
}
```

Allow me to walk you through this script. First of all, an `objlinkarrow` object will be created at the `x` and `y` positions defined by the first two arguments, respectively. Once this is created, execution of the script will pause in order for the new `objlinkarrow`'s Create event to execute. Once that is complete, the script can set the variables `creator`, `direction`, and `flinch_direction` appropriately, erasing any garbage values assigned to them prior during the Create event.

Notice that I chose not to comment this script. In my opinion, most of the commenting could be left in the Create event of the object. Here, all functions associated with the object could have their prototypes listed. This makes accessing the parameter list much easier because you are still within the object. Once in an object window, it is difficult to navigate away to the scripts section because Game Maker won't allow you to have a coding window and a script window open simultaneously. Additionally within the Create event, we could define all data members and initialize them as fits. It is good style to initialize all your variables to start with even if they have garbage values. This reminds whoever is working with the code the types of values they contain. (This is especially true in GML, which is not a strongly typed language.) The Create event might look something like this:

```
/*****
Name: objlinkarrow
Description: Link's arrow
Data members:
    instance_id creator; Instance ID of creator
    bool direction; Direction of arrow
    int flinch_direction; Direction (in degrees) target will flinch
    int h_speed; Horizontal speed of arrow
    int ttl; Number of steps before arrow is destroyed
    int damage; Amount of damage this inflicts
    int distance; Distance target will be sent
Function Prototypes:
    objlinkarrow constr_objlinkarrow(int x-position, int y-position,
                                     instance_id id, bool direction);
*****/

creator = noone;
direction = -1;
flinch_direction = -1;
h_speed = 4;
ttl = 25;
damage = 10;
distance = 5;
```

Observe the dummy values given to `creator`, `direction`, and `flinch_direction`. This poses no problem because, as stated before, these variables will be immediately changed by the constructor. It gives more information about them by showing foremost that they exist and secondly what data type they hold. Setting `creator` to equal the built in `noone` signals that this variable will hold the id of an instance, namely the id of the instance that called its constructor.

A major class design goal is limiting the interactions a given class has with other classes. This is a concept known as “loose-coupling.” It might be tempting to associate this arrow with `objlink` by writing in its description “Arrow created by `objlink`.” This implies a binding to `objlink`, which certainly doesn’t do Kirby any justice!

You may be wondering why we didn’t go ahead and initialize all of the variables in the constructor. A number of reasons make this a bad idea. First off, we have a Create event, and we shouldn’t completely ignore it. Secondly, we want to hide as much implementation detail as possible.

We now look at an example implementation of the constructor for `objlinkarrow`. Upon pressing the special attack button, Link would go into his bow and arrow pose. On Animation End, the code would look something like this

```
switch (sprite_index)
{
    case sprlinkxr:
        // create arrow
        constr_objlinkarrow(bbox_right, bbox_top + 32, id, face_right);
        break;
}
```

Here, `face_right` is a flag set in `objlink` that designates whether he is facing left (0) or right (1). Consequently, this same direction will get passed into the arrow.

### Inheritance

Inheritance is a key component of object oriented programming. Lucky for us, inheritance has some support in Game Maker. While not as powerful as C++’s inheritance, Game Maker’s inheritance provides a great method for reusing code and simplifying design.

In Smash Bros. Classic v3.0, one major new development that uses inheritance will be the enemy system. A base class called `objenemy` will be rooted in the inheritance hierarchy. This object will take care of elements that are familiar to all enemy objects, be it a Goomba or a Stalfos. All enemies will have a health, for example. So instead of defining a health variable in each and every enemy, we can define it in the base class and set up a chain of inheritance that guarantees, e.g., that our friend, the Waddle-Doo, gets a health. The next level of the enemy tree might be something more specific to enemies in a single game. To take advantage of this commonality, it would make sense to define, say, an `objzeldaenemy` object. One element that would be common to all Zelda enemies would be the explosion that they leave when they die. This affect could be programmed into the Destroy event of `objzeldaenemy`. Note also that we would never instantiate these objects (they wouldn't even have sprites!). They act much like an abstract class in C++ or Java.

Game Maker is event based, so code can be inherited under each event. If the parent class has something defined in its Create event, its child class will also get that behavior *as long as it has nothing defined in its Create event*. Actually, that's not the whole truth. Game Maker provides the function `event_inherited()` that will run the parent class' behavior in addition to the child's unique behavior. This comes with a caveat if your code depends on this kind of structure:

```
if (var == true)
{
    exit;
}
```

Game Maker requires that you include this in the parent and the child to achieve the desired behavior. Exit only gets out of the current code block, but when we are inheriting via `event_inherited`, we are actually inheriting a whole different code block. Here's a quick example. Say that my `objfighter` base, abstract class has the following Step event:

```
if(flag_flinch == true || flag_taunt == true)
{
    exit;
}

if(place_free(x, y + 1))
{
    gravity = gravity_force;
    ...
}
```

```
else
{
...
}
...
```

Now, we create an `objluigi` object that inherits from the `objfighter` class. `objluigi` has a huge amount of unique behavior associated with it. We want the same basic behavior found in `objfighter`, and we still don't want to execute either's behavior if the `flag_flinch` or the `flag_taunt` variables are true. So we design `objluigi`'s Step event as follows:

```
event_inherited();

if(keyboard_check_pressed(global.key_special) == true)
{
// create some green fireworks
...
}
...
```

So this would be perfectly acceptable, right? Well, not exactly. `event_inherited()` does a little more than copy the text from the inherited event into the current event. What you will find happening is that even if Luigi is taunting, he will be able to do his special attack! The `exit` statement is only associated with the block of code from within the base class, `objfighter`, so exiting from it will not exit out of `objluigi`'s block of code. We would be preventing things such as gravity from occurring, but everything after `event_inherited()` works as normal. To fix this we would either never use `exit` in this way or copy the `exit` construct after `event_inherited()`.

### Tools

#### Game Maker 7.0

A program for creating computer games. See <http://www.gamemaker.nl> for more information.

#### SVN

The subversion utility is a version control system. It provides a means for organizing team work and distributing updates. It requires a central server. Such a server can be

implemented for free at <http://www.assembla.com>. It was developed as a replacement for CVS. TortoiseSVN is a windows-based SVN client.

### **Audacity**

This is a great sound editor, and it is freeware!

### **Notepad++**

This is a free text-editor for Windows with syntax highlighting and tabs. It makes a great website source code manager.

---

<sup>1</sup> One of the most influential pieces of computer literature for me was a book by Steven Levy called [Hackers: Heroes of the Computer Revolution](#).

<sup>2</sup> Global Variables. Wikipedia. October 2007.

<sup>3</sup> Delphi, what GameMaker is interpreted with, suggests the use of ‘ ‘ (single quotes) for strings. This convention is “against the grain,” and I’ve suggested a convention more commonly seen (C, C++, Java).