Module Guide for Chipmunk2D Game Physics Library

Alex Halliwushka and Luthfi Mawarid

June 1, 2018

Contents

1 Introduction

Decomposing a system into modules is a commonly accepted approach to developing software. A module is a work assignment for a programmer or programming team (Parnas et al., 1984). In the best practices for scientific computing, Wilson et al. (2013) advise a modular design, but are silent on the criteria to use to decompose the software into modules. We advocate a decomposition based on the principle of information hiding (Parnas, 1972). This principle supports design for change, because the "secrets" that each module hides represent likely future changes. Design for change is valuable in SC, where modifications are frequent, especially during initial development as the solution space is explored.

Our design follows the rules laid out by Parnas et al. (1984), as follows:

- System details that are likely to change independently should be the secrets of separate modules.
- Each data structure is used in only one module.
- Any other program that requires information stored in a module's data structures must obtain it by calling access programs belonging to that module.

After completing the first stage of the design, the Software Requirements Specification (SRS), the Module Guide (MG) is developed (Parnas et al., 1984). The MG specifies the modular structure of the system and is intended to allow both designers and maintainers to easily identify the parts of the software. The potential readers of this document are as follows:

- New project members: This document can be a guide for a new project member to easily understand the overall structure and quickly find the relevant modules they are searching for.
- Maintainers: The hierarchical structure of the module guide improves the maintainers' understanding when they need to make changes to the system. It is important for a maintainer to update the relevant sections of the document after changes have been made.
- Designers: Once the module guide has been written, it can be used to check for consistency, feasibility and flexibility. Designers can verify the system in various ways, such as consistency among modules, feasibility of the decomposition, and flexibility of the design.

The rest of the document is organized as follows. Section 2 lists the anticipated and unlikely changes of the software requirements. Section 3 summarizes the module decomposition that

was constructed according to the likely changes. Section 4 specifies the connections between the software requirements and the modules. Section 5 gives a detailed description of the modules. Section 6 includes two traceability matrices. One checks the completeness of the design against the requirements provided in the SRS. The other shows the relation between anticipated changes and the modules. Section 7 describes the use relation between modules.

2 Anticipated and Unlikely Changes

This section lists possible changes to the system. According to the likeliness of the change, the possible changes are classified into two categories. Anticipated changes are listed in Section 2.1, and unlikely changes are listed in Section 2.2.

2.1 Anticipated Changes

Anticipated changes are the source of the information that is to be hidden inside the modules. Ideally, changing one of the anticipated changes will only require changing the one module that hides the associated decision. The approach adapted here is called design for change. Anticipated changes are numbered by **AC** followed by a number.

AC1: The specific hardware on which the software is running.

AC2: The data structure of the physical properties of an object such as the object's mass, position and velocity.

AC3: The data structure of the surface properties of an object such as the object's friction and elasticity.

AC4: How all the rigid bodies and shapes interact together.

AC5: The data structure containing collision information such as the objects that collide and their mass.

AC6: How the overall control of the simulation is orchestrated, including the input and output of data.

AC7: The implementation of mathematical vectors.

AC8: The implementation of bounding box structures.

AC9: The implementation of transformation matrices.

AC10: How the simulation space is spatially indexed.

AC11: The algorithm used for solving collisions.

AC12: The implementation for the sequence (array) data structure.

AC13: The implementation for the linked (tree) data structure.

AC14: The implementation for the associative (hash table) data structure.

2.2 Unlikely Changes

The module design should be as general as possible. However, a general system is more complex. Sometimes this complexity is not necessary. Fixing some design decisions at the system architecture stage can simplify the software design. If these decision should later need to be changed, then many parts of the design will potentially need to be modified. Hence, it is not intended that these decisions will be changed. Unlikely changes are numbered by **UC** followed by a number.

UC1: Input/Output devices (Input: Keyboard and/or mouse, Output: Memory and/or Screen).

UC2: There will always be a source of input data external to the software.

UC3: Output data are displayed to the output device.

UC4: The goal of the system is to simulate the interactions of 2D rigid bodies.

UC5: A Cartesian coordinate system is used.

UC6: All objects are rigid bodies.

UC7: All objects are 2D.

3 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets in Table 1. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented. Modules are numbered by **M** followed by a number.

M1: Hardware-Hiding Module

M2: Rigid Body Module

M3: Shape Module

M4: Circle Module

M5: Segment Module

M6: Polygon Module

M7: Space Module

M8: Arbiter Module

M9: Control Module

M10: Vector Module

M11: Bounding Box Module

M12: Transform Matrix Module

M13: Spatial Index Module

M14: Collision Solver Module

M15: Sequence Data Structure Module

M16: Linked Data Structure Module

M17: Associative Data Structure Module

Note that M1 is a commonly used module and is already implemented by the operating system. It will not be reimplemented.

Level 1	**Level 2**	**Level 3**
Hardware-Hiding Module		
Behaviour-Hiding Module	Rigid Body Module	
	Shape Module	Circle Module Segment Module Polygon Module
	Space Module	
	Arbiter Module	
	Control Module	
Software Decision Module	Vector Module	
	Transform Matrix Module	
	Bounding Box Module	
	Spatial Index Module	
	Collision Solver Module	
	Sequence Data Structure Module	
	Linked Data Structure Module	
	Associative Data Structure Module	

Table 1: Module Hierarchy

4 Connection Between Requirements and Design

The design of the system is intended to satisfy the requirements developed in the SRS. In this stage, the system is decomposed into modules. The connection between requirements and modules is listed in Table 2.

5 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by Parnas et al. (1984). The *Secrets* field in a module decomposition is a brief statement of the design decision hidden by the module. The *Services* field specifies *what* the module will do without documenting *how* to do it. For each module, a suggestion for the implementing software is given under the *Implemented By* title. If the entry is *OS*, this means that the module is provided by the operating system or by standard programming language libraries. If the entry is *Chipmunk*, this means that the module will be implemented by the game physics library. If a dash (–) is shown, this means that the module is not a leaf and will not have to be implemented. Whether or not this module is implemented depends on the programming language selected. If a module has a children section, the modules listed will inherit from this module. Both the parent and the children modules will be implemented.

5.1 Hardware Hiding Modules (M1)

Secrets: The data structure and algorithm used to implement the virtual hardware.

Services: Serves as a virtual hardware used by the rest of the system. This module provides the interface between the hardware and the software. So, the system can use it to display outputs or to accept inputs.

Implemented By: OS

5.2 Behaviour-Hiding Module

Secrets: The contents of the required behaviours.

Services: Includes programs that provide externally visible behavior of the system as specified in the software requirements specification (SRS) documents. This module serves as a communication layer between the hardware-hiding module and the software decision module. The programs in this module will need to change if there are changes in the SRS.

Implemented By: –

5.2.1 Rigid Body Module (M2)

Secrets: The data structure of a rigid body.

Services: Stores the physical properties of an object, such as mass, position, rotation, velocity, etc, and provides operations on rigid bodies, such as setting the mass and velocity of the body.

Implemented By: Chipmunk

5.2.2 Shape Module (M3)

Secrets: The data structure of a collision shape.

Services: Stores the surface properties of an object, such as friction or elasticity, and provides operations on shapes, such as setting its friction or elasticity.

Children: Circle Module, Segment Module, Polygon Module

Implemented By: Chipmunk

Circle Module (M4)

Secrets: The data structure of a circle shape.

Services: Provides operations on circles such as initializing a new circle, calculating moment and area, etc.

Implemented By: Chipmunk

Segment Module (M5)

Secrets: The data structure of a segment shape.

Services: Provides operations on segments such as initializing a new segment, calculating moment and area, etc.

Implemented By: Chipmunk

Polygon Module (M6)

Secrets: The data structure of a polygon shape.

Services: Provides operations on polygons such as initializing a new polygon, calculating moment, area and centroid, etc.

Implemented By: Chipmunk

5.2.3 Space Module (M7)

Secrets: The container for simulating objects.

Services: Controls how all the rigid bodies and shapes interact together.

Implemented By: Chipmunk

5.2.4 Arbiter Module (M8)

Secrets: The data structure containing collision information.

Services: Stores all collision data, such as which bodies collided and their masses.

Implemented By: Chipmunk

5.2.5 Control Module (M9)

Secrets: The algorithm for coordinating the running of the program and the interface for receiving inputs and sending outputs.

Services: Provides the main program.

Implemented By: Chipmunk

5.3 Software Decision Module

Secrets: The design decision based on mathematical theorems, physical facts, or programming considerations. The secrets of this module are *not* described in the SRS.

Services: Includes data structure and algorithms used in the system that do not provide direct interaction with the user.

Implemented By: –

5.3.1 Vector Module (M10)

Secrets: The data structure representing vectors.

Services: Provides vector operations such as addition, scalar and vector multiplication, dot and cross products, rotations, etc.

Implemented By: Chipmunk

5.3.2 Bounding Box Module (M11)

Secrets: The data structure for representing bounding boxes.

Services: Provides constructors for bounding boxes and operations such as merging boxes, calculating their centroids and areas, etc.

Implemented By: Chipmunk

5.3.3 Transform Matrix Module (M12)

Secrets: The data structure representing transformation matrices.

Services: Provides constructors for affine transformation matrices, matrix operations such as inverse, transpose, multiplications, and operations for applying transformations to vectors and bounding boxes.

Implemented By: Chipmunk

5.3.4 Spatial Index Module (M13)

Secrets: The data structures and algorithms for spatially indexing the simulation space.

Services: Provides spatial indexing operations and tracks the positions of bodies in the simulation space.

Implemented By: Chipmunk

5.3.5 Collision Solver Module (M14)

Secrets: The data structures and algorithms for detecting collisions.

Services: Fast collision filtering, primitive shape to shape collision detection.

Implemented By: Chipmunk

5.3.6 Sequence Data Structure Module (M15)

Secrets: The data structure for a sequence data type.

Services: Provides array manipulation operations, such as building an array, accessing a specific entry, slicing an array, etc.

Implemented By: Chipmunk

5.3.7 Linked Data Structure Module (M16)

Secrets: The data structure for a linked data type.

Services: Provides tree manipulation operations, such as building a tree, accessing a specific entry, etc.

Implemented By: Chipmunk

5.3.8 Associative Data Structure Module (M17)

Secrets: The data structure for an associative data type.

Services: Provides operations on hash tables, such as building a hash table, accessing a specific entry, etc.

Implemented By: Chipmunk

6 Traceability Matrix

This section shows two traceability matrices: between the modules and the requirements, and between the modules and the anticipated changes. We should also consider documenting the mapping between these "abstract" modules and the implemented code files.

Req.	**Modules**
R1	M7, M9, M15
R2	M2, M9, M10, M12
R3	M3, M4, M5, M6, M9, M10
R4	M2, M3, M4, M5, M6, M7, M9
R5	M2, M7, M10, M12
R6	M2, M7, M10, M12
R7	M2, M7, M11, M13, M14, M16, M17
R8	M2, M7, M10, M12, M8

Table 2: Trace Between Requirements and Modules

AC	**Modules**
AC1	M1
AC2	M2
AC3	M3, M4, M5, M6
AC4	M7
AC5	M8
AC6	M9
AC7	M10
AC8	M11
AC9	M12
AC10	M13
AC11	M14
AC12	M15
AC13	M16
AC14	M17

Table 3: Trace Between Anticipated Changes and Modules

7 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. Parnas (1978) said of two programs A and B that A *uses* B if correct execution of B may be necessary for A to complete the task described in its specification. That is, A *uses* B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B. Figure 1 illustrates the use relation between the modules. It can be seen that the graph is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system, and modules in the higher level of the hierarchy are essentially simpler because they use modules from the lower levels. As the modules M10, M11 and M12 are used by many other modules, their uses have been color-coded for clarity.

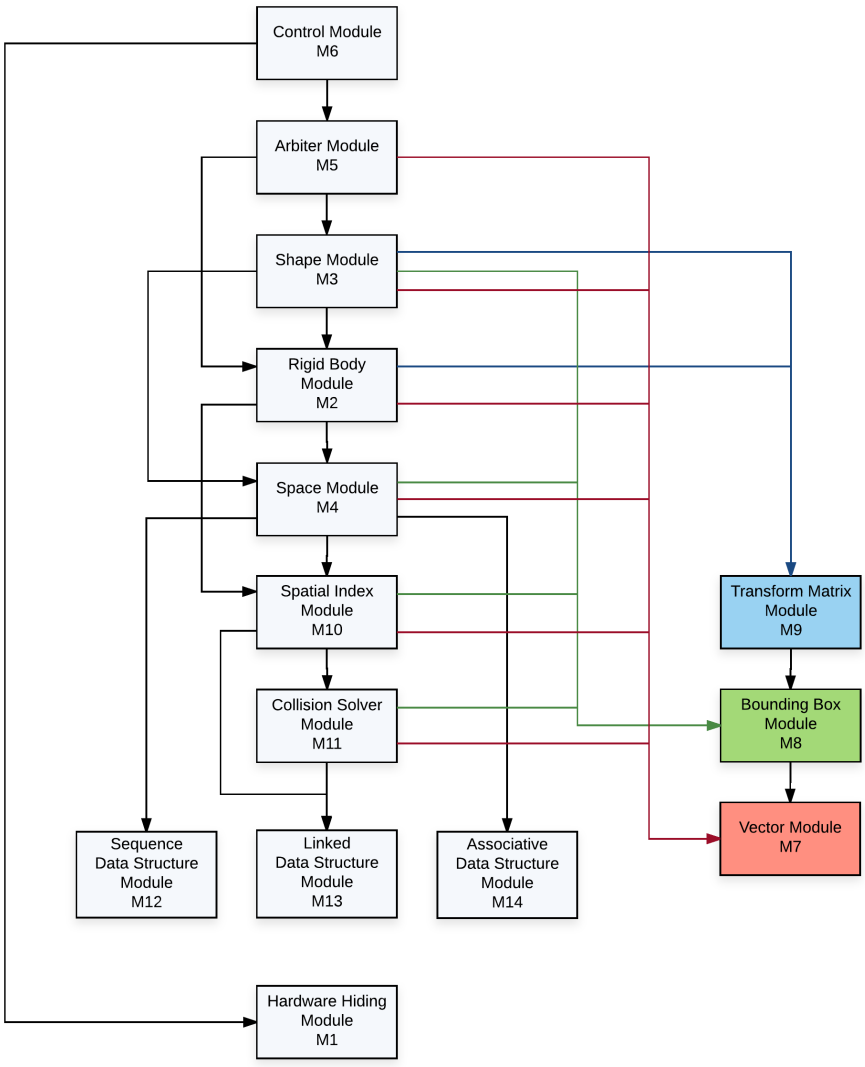

Figure 1: Use hierarchy among Modules

8 Inheritance Hierarchy Between Modules

In this section the inheritance hierarchy between modules is provided (Figure 2). Modules in the lower level of the hierarchy inherit properties from the higher levels.

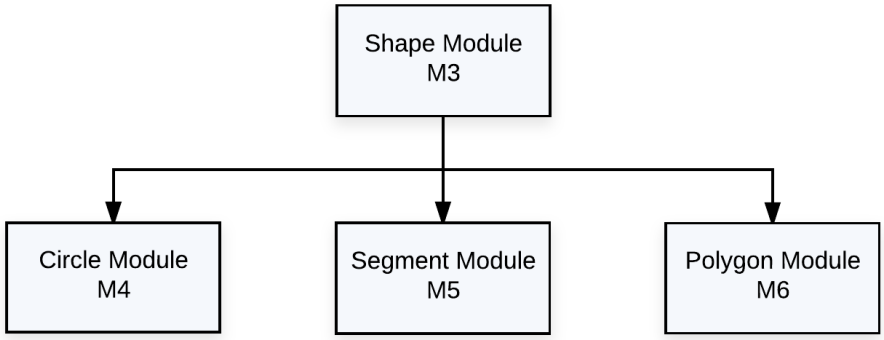

Figure 2: Inheritance hierarchy among Modules

References

D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, pages 408–417, Piscataway, NJ, USA, 1984. IEEE Press. ISBN 0-8186-0528-6.

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM, vol. 15, no. 2, pp. 1053-1058*, December 1972.

David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press. ISBN none.

Greg Wilson, D.A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H.D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumblet, Ben Waugh, Ethan P. White, and Paul Wilson. Best practices for scientific computing. *CoRR*, abs/1210.0530, 2013.