Module Guide for Open Source Game Physics Library

Alex Halliwushka

August 24, 2015

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 (**?**). In the best practices for scientific computing, **?** 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 (**?**). 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 **?**, 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 (**?**). 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 is 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.

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 elastiicity

AC4: The data structure of the collision such as the objects that collide and their mass.

AC5: The data structure of how the bodies are constrained, such as the bodies that are constrained and the type of constraint.

AC6: How all the rigid bodies, shapes, and constraints interact together

AC7: How the overall control of the simulation is orchestrated.

AC8: The implementation for the sequence (array) data structure.

AC9: The implementation for the linked (tree) data structure.

AC10: The implementation for the associative (hash table) data structure.

AC11: The algorithm used for solving collisions.

AC12: The algorithm used for spatial queries.

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.

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

UC2: Output data are displayed to the output device.

UC3: The goal of the system is to simulate the interactions of 2D rigid bodies

UC4: A Cartesian coordinate system is used

UC5: All objects are rigid bodies

UC6: 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.

M1: Hardware-Hiding Module (Hardware)

M2: Rigid Body Module (Rigid)

M3: Shape Module (Shape)

M4: Arbiter Module (Arbiter)

M5: Constraint Module (Constraint)

M6: Space Module (Space)

M7: Control Module (Control)

M8: Sequence Data Structure Module (Sequence)

M9: Linked Data Structure Module (Linked)

M10: Associative Data Structure Module (Associative)

M11: Collision Solver Module (Collision)

M12: Query Module (Query)

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
	Arbiter Module	
	Constraint Module	Slide Joint Module Ratchet Joint Module Rotary Joint Module Pivot Joint Module Pin Joint Module Groove Joint Module Gear Joint Module Damped Spring Module Rotary Spring Module
	Space Module	
	Control Module	
	Collision Solver Module	
	Query Module	
Software Decision 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 **?**. 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. The operations on rigid bodies such as setting the mass and velocity of the object are also included in this module.

Implemented By: Chipmunk

5.2.2 Shape Module (M3)

Secrets: The data structure of a shape.

Services: Stores the surface properties of an object such as friction or elasticity. The operation on shapes such as setting the friction are also included in this module.

Children: Circle Module, Segment Module, Polygon Module

Implemented By: Chipmunk

5.2.3 Arbiter Module (M4)

Secrets: The data structure of a collision

Services: Stores all the data about the collision such as which bodies collided and their mass.

Implemented By: Chipmunk

5.2.4 Constraint Module (M5)

Secrets: Data structure of a constraint

Services: Stores the type of constraint between the two bodies and how it affects the bodies.

Children: Slide Joint Module, Ratchet Joint Module, Rotary Joint Module, Pivot Joint Module, Pin Joint Module, Groove Joint Module, Gear Joint Module, Gear Joint Module, Damped Spring Module, Rotary Spring Module

Implemented By: Chipmunk

5.2.5 Space Module (M6)

Secrets: The container for simulating objects.

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

Implemented By: Chipmunk

5.2.6 Control Module (M7)

Secrets: The algorithm for coordinating the running of the program.

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 Sequence Data Structure Module (M8)

Secrets: The data structure for a sequence data type.

Services: Provides array manipulation, including building an array, accessing a specific entry, slicing an array etc.

Implemented By: Chipmunk

5.3.2 Linked Data Structure Module (M8)

Secrets: The data structure for a Linked data type.

Services: Provides tree manipulation, including building a tree, accessing a specific entry etc.

Implemented By: Chipmunk

5.3.3 Associative Data Structure Module (M8)

Secrets: The data structure for a associative data type.

Services: Provides array manipulation, including building a hash table, accessing a specific entry etc.

Implemented By: Chipmunk

5.3.4 Collision Solver Module (M11)

Secrets: The data structures and algorithms for detecting collisions.

Services: Spatial Indexing, fast collision filtering, constraint based filtering, primitive shape to shape collision detection.

Implemented By: Chipmunk

5.3.5 Query Module (M12)

Secrets: The data structures and algorithms for spatial queries

Services: Nearest point queries, segment queries, shape queries, fast bounding box queries.

Implemented By: Chipmunk

6 Traceability Matrix

This section shows a traceability matrix: between the modules and the requirements.

Req.	Modules
R1	Space, Control,Sequence
R2	Rigid, Control
R3	Shape, Control
R4	Constraint, Control
R5	Control
R6	Rigid, Space
R7	Rigid, Space, Collision
R8	Rigid, Arbiter
R9	Rigid
R10	Rigid, Constraint, Linked, Associative
R11	Rigid, Space, Linked, Associative, Query

Table 2: Trace Between Requirements and Modules

7 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. **?** 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.

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.

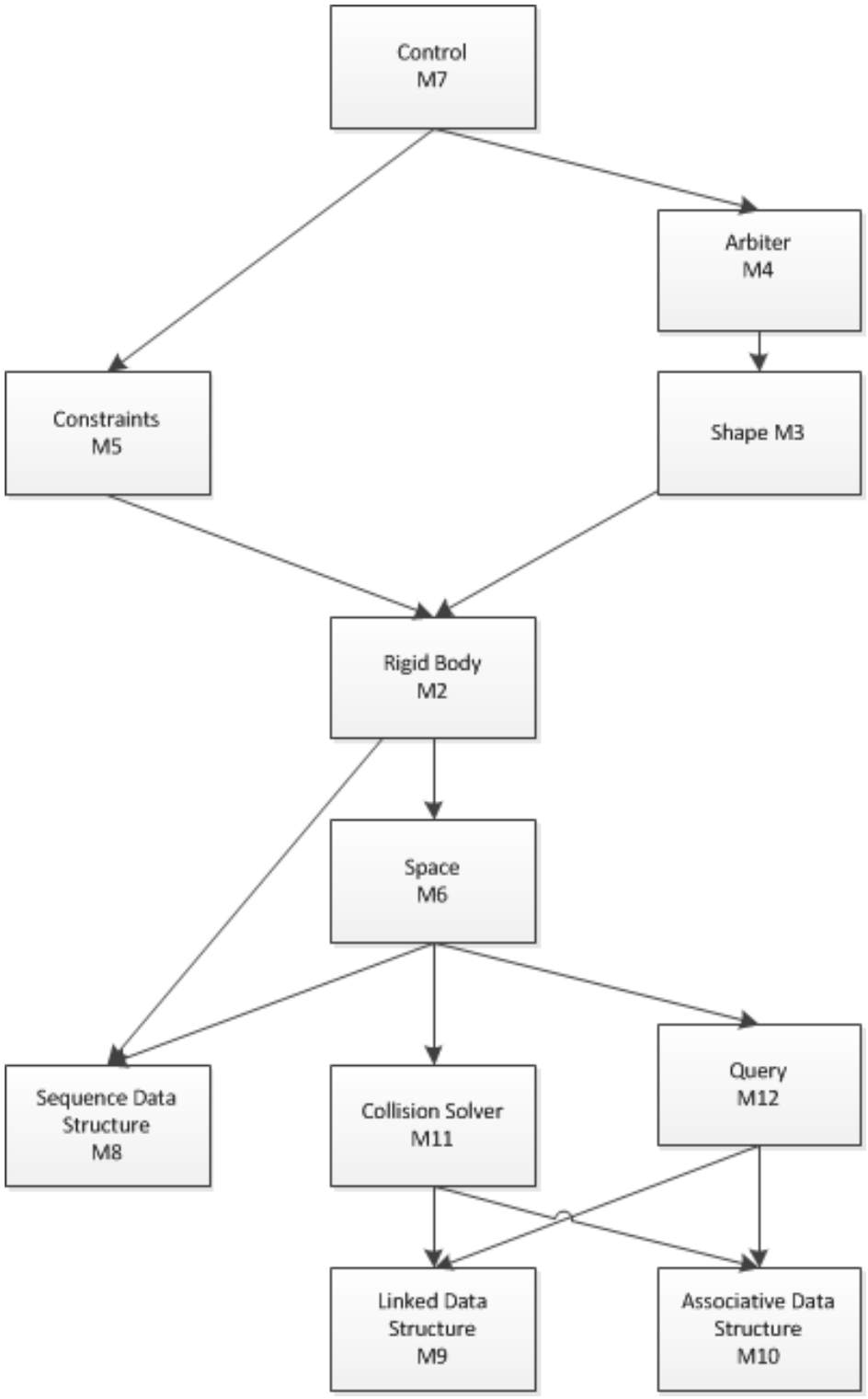

Figure 1: Use hierarchy among Modules

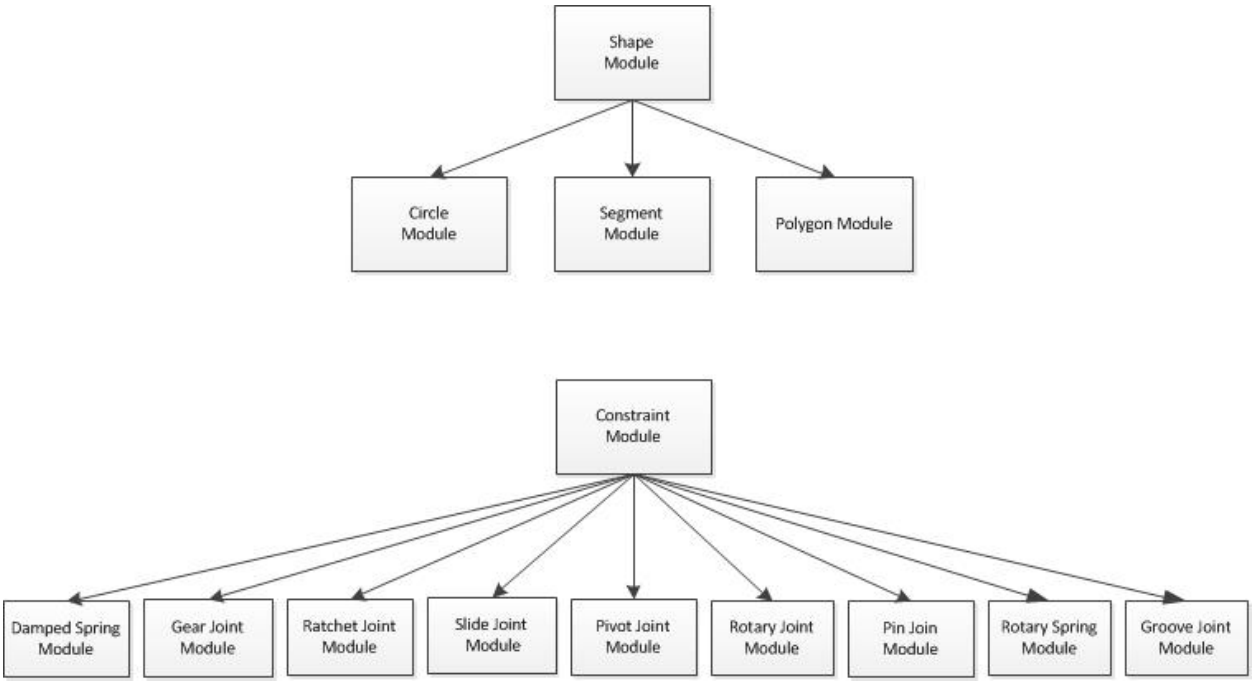

Figure 2: Inheritance hierarchy among Modules