Ruleset Development with tpserver-cpp

Lee Begg

April 28, 2008

Contents

Abstract

The document outlines the process for developing rulesets for tpserver-cpp. Since the Thousand Parsec project overall and tpserver-cpp in particular are developing fast, this is a living document, changing as the tpserver-cpp environment changes.

Developing a ruleset is not a trivial process, so this document is designed to guide you through the steps you need to do and highlight issues and probable sticking points. Hopefully it make ruleset development easy enough that several parallel ruleset developments can take place.

Part I

The Big Picture

Chapter 1

Thousand Parsec

Started in 2002 by mithro (Tim Ansell) and llnz (Lee Begg), at mithro's instigation, the project was aiming to become a framework for building online turn base space strategy games. Thousand Parsec was originally conceived as a "Stars!" clone, but taking inspiration from the WorldForge project—building a framework for creating massively multiplayer online role playing games—that both mithro and llnz worked on, it was decided to go for the much more general goal.

The framework can be divided into five areas.

- Protocol
- Servers
- Clients
- Media
- Tools

The protocol is described fully in chapter 2. Servers provide games using the TP protocol, and can do it in different ways. They can provide multiple games at once, and can host different games. Clients provide the user's view of the game. A client can connect to any server, and users should be able to use multiple different clients to connect to any game. The media collected by the project can be used by any game. Tools are used to develop rulesets and setup/manage server and other useful tasks.

The whole framework is under development.

Chapter 2

Protocol

The Thousand Parsec protocol is the truly defining work of Thousand Parsec. It sets the common understanding between servers and clients. If a game can work within the restrictions of the protocol then the whole Thousand Parsec framework can be used.

Chapter 3

Introduction to tpserver-cpp

tpserver-cpp is the software written in C++ for Thousand Parsec which implements a Thousand Parsec protocol Server. It was mostly designed and written by Lee Begg (llnz). In it's early days, Lee used it as the project for his Honours Level (4th year university) Software Engineering project.

It supports only one game at a time. It uses plugins (dynamically loaded libraries) to implement tpscheme (tpcl), persistence methods, and rulesets. It is single threaded, using non-blocking sockets and an event-style processing model.

tpserver-cpp has a nice console terminal, using libtprl. Underneath, libtprl is a wrapper around Readline, so all readline editing features, and history, are available.

Chapter 4

Tools Used

Thousand Parsec use a number of tools in creating it's framework. Some of those specific to tpserver-cpp are outlined below.

4.1 Git/Cogito

Git is a distributed source revision control system. This allow many people to work simultaneously, even disconnected from the internet. Changes are recorded in snapshots, and are sent and push around. As it is used by the Linux Kernel, it is developing and well supported.

A slightly nicer interface to git is Cogito, with commands starting with "*cg-*".

4.2 Autotools

The collection of aclocal, autoheader, autoconf, and automake make up "autotools". They are mostly used with C and C++ based projects, providing a slightly nicer way of configuring the build environment and build system than by scratch (in for example, sh/m4 and Makefile).

A replacement for autotools is not out of the question because of the complexity of using autotools and some platforms are not well supported.

4.2.1 Libtool

Libtool is also consided part of autotools. It provides a generic way to create libraries and dynamic module loading programs.

Because libtool was so wonderful in creating loadable libraries on many platforms and the lack of a common way to load said libraries, libtool has a companion library called libltdl—the libtool Dynamic Loader. It has an interface similar to `dlopen()` that works on all the systems libtool does.

This includes systems that don't actually dynamically load, so libtool and libltdl have hacks to allow what looks like dynamic loading, but isn't.

Chapter 5

Licensing

This chapter is yet to be written, but will mention the GPL, the server hole, and how things get confusing with loadable modules.

Part II

Starting out

Chapter 6

Building tpserver-cpp

README, INSTALL, man page.

tpserver-cpp uses libtprl for the console in the server. Only libtprl is required to build tpserver-cpp, all other packages mentioned are optional.

To build tpserver-cpp, first you will need libtprl installed. If you distribution has a high enough version of the library, you should install that. Otherwise you will need to build from source.

With libtprl installed, we can now build tpserver-cpp itself. The first set is to get the source for the server, either from a release, or from the bleeding edge git repository.

If you choose to start with a released version of tpserver-cpp, download the tarball (*tar.gz* files), and then you can run the following commands:

```
>$ tar xzf tpserver-cpp-<version>.tar.gz
>$ cd tpserver-cpp-<version>
```

If you chose to build the current bleeding edge from git, you need to clone the repository and set up the build environment.

```
>$ cg-clone git://git.thousandparsec.net/git/tpserver-cpp.git
>$ cd tpserver-cpp
```

This downloads the source from the git repo. Then the first time, and any time a *.in*, *.ac* or *.am* file changes, you need to run *autogen.sh* to generate the configure script, and create the build system.

```
>$ ./autogen.sh
```

Now both the released tarball and the bleeding edge from git are ready to be configured and built.

The configure script takes a number of options and settings. Running `./configure` with the `--help` option prints out information about the various options. Some of the important options are highlighted in *README*.

Some environmental variables are also recorded by configure, among these is `PKG_CONFIG_PATH` which can be used to help the *configure* script find libtprl. For example, if you prefix for libtprl is */opt/tp/*, the you should run `PKG_CONFIG_PATH=/opt/tp/lib/pkg_config ./configure`.

```
>$ ./configure
>$ make
>$ make install
```

Chapter 7

Configuring tpserver-cpp

Once tpserver-cpp is built and installed, you can run tpserver-cpp. You won't see much though as the default configuration is very basic.

The settings for the server can be specified on the command line, read from file, or entered on the console. The settings are recorded in that order, so the file overwrites the command line for example.

All the settings are defined and described in the *sample.conf* files. Some modules have the own *sample.conf* files as well.

Chapter 8

Basic Design

The tpserver-cpp source tree is divided into two parts. The first part, "*tpserver*" is the core of the server and implements all the common parts of the server. The second part is the "*modules*" tree.

Also in the root directory is a copy of libltdl because some distributions (namely debian and ubuntu) ship broken versions.

8.1 *tpserver* tree

This tree contains the bulk of the core server.

8.2 *modules* tree

The *modules* tree contains the modules the server ships with. They are divided by the type of module they are, namely "*game*" for rulesets, "*persistence*" for persistence methods, and "*tpcl*" for scheme (tpcl) implementations.

Part III

Ruleset Development

Chapter 9

The Ruleset Concept

In tpserver-cpp, rulesets define how the game behaves. All the properties and rules which define the game are in the ruleset.

The core server implements all the things that are common between rulesets. This includes how to talk to clients, how to cache the data between clients and persistence, and what the various data types look like.

Chapter 10

Ruleset parts in tpserver-cpp

The development of a ruleset can be broken down into parts. Some of the parts are related to a single class, where others touch on many classes.

10.1 `Ruleset` Class

The `Ruleset` class is the "public face" of the ruleset. It handles a few tasks relating to the initialisation, creating and running the game, and players.

Two functions, `getName()` and `getVersion()` are used to provide the name and version of the ruleset, and is used to publish the ruleset name to the metaserver and to DNS-SD on the local network.

The `initGame()` method sets the server up ready for the game to be created or resumed from persistence.

The `createGame()` creates everything to start the game with. This includes creating the universe, initialising all the components and properties, and anything else that only has to be done once so the game can be played.

The method `startGame()` is called when the game is made active and in progress, including everytime the server is started with the game already created and in persistence.

Two functions handle the creation of player for the ruleset. The function `onPlayerAdd()` allows the ruleset to decide if the player is allowed to be added. Most of the time they will be allowed, but if there is limited number of players then the ruleset can say no to creating the new player. The method `onPlayerAdded()` allows the ruleset to do things—such as give the player a home planet—now that the player has been added to the game.

10.2 `TurnProcess` Class

The `TurnProcess` class is used to perform the tasks required to end the turn. As such, the virtual method called is `doTurn()`. Other methods might be provided to subclasses in future to simplify and optimise the turn processing.

10.3 Object Types

The object types define the properties of objects that appear in the universe.

10.4 Order Types

Order types define what actions objects can take.

In the End of Turn process, the method `doOrder()` is called, where the actual work is done.

10.5 Categories, Designs, Components and Properties

Chapter 11

Where to Begin

A ruleset normally starts simply with an idea. This could be wanting to implement an existing game in any format, or it could be a concept for an all new game.

The next step is to plan out as much of the game as possible. Gathering or creating as much information as possible about what is in the game helps later on. Information includes: what object types, what orders can be given to those objects, what order things a processed at the end of turn, what types of designs can be made, what components those designs are made of, and what properties are needed for those designs and components. Also needed is how combat will be resolved if the game includes it. Another important element of any idea is the name that the ruleset will be called.

The third step is to code, and the fourth is test, followed by more coding and testing.

Chapter 12

Setting up the Build System

The build system is the process used to create the ruleset module from the source code. The method used varies if you build your module in the tpserver-cpp source tree, or you build it in its own separate source tree.

12.1 In Tree Modules

In tree modules, such as the provided Minisec and MTSec rulesets, use the tpserver-cpp build system. This system is build on Autoconf and Automake. The first step is to make a directory in the *modules/games/* directory to hold the source code for the module.

```
>$ cd modules/games
>$ mkdir mygame
```

Next, you need to create a *Makefile.am* file in the new directory. It's contents should be something like:

```
rulesetlib_LTLIBRARIES = libmygame.la

rulesetlibdir = $(libdir)/tpserver/ruleset

libmygame_la_SOURCES = mygame.cpp

libmygame_la_LDFLAGS = -module

noinst_HEADERS = mygame.h
```

As you add more source files, you add the *.cpp* files to `SOURCES`, and the header files to `noinst_HEADERS`. You can break long lines onto multiple lines by putting a backslash (\) at the end of the line and continuing on the next

line. Note that the `SOURCES` and `LDFLAGS` variables start with the name of the library with underscore instead of the fullstop.

Lastly, we need the build system to actually process the new directory with it's *Makefile.am* file. Two files need to be change. The first is to add the new directory to the `SUBDIRS` of *modules/games/Makefile.am*, such as:

```
SUBDIRS = mtsec minisec mygame
```

The second file that needs changing is *configure.ac*. This file needs to know about the *Makefile.am* file to make it into first *Makefile.in* and then a normal *Makefile*. We add a line about our new *Makefile.am* in the `AC_CONFIG_FILES` section that is right at the end of *configure.ac* file:

```
AC_CONFIG_FILES([
Makefile
tpserver/Makefile
modules/Makefile
modules/games/Makefile
modules/games/minisec/Makefile
modules/games/mtsec/Makefile
modules/persistence/Makefile
modules/persistence/mysql/Makefile
modules/tpcl/Makefile
modules/tpcl/guile/Makefile
modules/tpcl/mzscheme/Makefile
modules/games/mygame/Makefile
])
```

Now the build system is ready for the code for the ruleset to be written. You will need to run `./autogen.sh` and `./configure` each time you edit your *Makefile.am*, so it would be a good idea to do that now. Note that because it specifies *mygame.cpp* as a source file and we haven't created it yet, the build system will not make the ruleset module.

12.2 Out of Tree Modules

You can use any build system you want if you are building out of tree. One point to keep in mind though is the location that the ruleset module needs to be installed to. It should match the location expected by tpserver-cpp. Typical locations could be one of the following:

- */usr/lib/tpserver/ruleset/*
- */usr/local/lib/tpserver/ruleset/*

- */opt/tp/lib/tpserver/ruleset/*
- *$PREFIX/lib/tpserver/ruleset/*

If the ruleset module isn't installed in the correct location, the name of the ruleset in the settings will need to include the path to the module. For example:

```
#Ruleset module installed in /usr/local/lib/mygamename/
#while server expects them in /usr/lib/tpserver/rulesets/

ruleset = "/usr/local/lib/mygamename/libmyruleset"

#compare with the following
#ruleset = myruleset
```

As can be seen, it is preferable to have the ruleset module installed in the right place to make configuring easier for game admins to set up games. But using the path can be handy in testing the ruleset without having to install it after every change.

Chapter 13

Writing the Ruleset

As mentioned in section 10.1, the Ruleset class is the main interface to the ruleset for the core server to use. So now it's time to start writing it.

Obviously, the Ruleset class for this ruleset module must be a subclass of Ruleset, and must implement all the pure virtual methods. So, the header (*mygame.h*) is straight forward. Full source code is in Appendix A.

```
#ifndef MYGAME_MYGAMERULESET_H
#define MYGAME_MYGAMERULESET_H
// License and a little documentation here

#include <tpserver/ruleset.h>

namespace MyGame{

class MyGameRuleset : public Ruleset{
 public:
  MyGameRuleset();
  virtual ~MyGameRuleset();

  std::string getName();
  std::string getVersion();
  void initGame();
  void createGame();
  void startGame();
  bool onAddPlayer(Player* player);
  void onPlayerAdded(Player* player);

};

} // namespace MyGame
```

```
#endif
```

Now, implementing the nine methods isn't too hard for a start. We will break it down and comment on this one step at a time. These would be written in the *mygame.cpp* file.

The first thing in the *mygame.cpp* file should be a one line comment saying what is in this file, and then the copyright and license for the file (see section 5). Following that, is where the include directives are. It helps maintainance if you keep them organised and in groups—for example; system includes, tpserver-cpp core includes, ruleset local includes, then the header file for this implementation file.

```
// License and a little documentation here

// System includes
#include <sstream>

// tpserver includes
#include <tpserver/game.h>
#include <tpserver/logging.h>
#include <tpserver/objectdatamanager.h>
#include <tpserver/ordermanager.h>
#include <tpserver/player.h>

// mygame includes

// header include
#include "mygame.h"
```

The tpserver includes above we will need later, and don't worry that the mygame includes section is empty as we will be adding to it soon.

After the includes is a good place for a little bit of libtool/libltdl magic, which we will come back to shortly.

The first methods are the constructor and destructor, and most of the time neither will do anything.

```
MyGameRuleset::MyGameRuleset(){
}

MyGameRuleset::~MyGameRuleset(){
}
```

The `getName()` and `getVersion()` methods both return the obvious strings.

```
std::string MyGameRuleset::getName(){
  return "MyGame";
}

std::string MyGameRuleset::getVersion(){
  return "0.1";
}
```

For a start, the next three methods dealing with initialisation, creation and starting are nearly empty, but will get filled in as more of the ruleset is written.

```
void MyGameRuleset::initGame(){
  Logger::getLogger()->info("MyGame initialised"};
}

void MyGameRuleset::createGame(){
  Logger::getLogger()->info("MyGame created"};
}

void MyGameRuleset::startGame(){
  Logger::getLogger()->info("MyGame started"};
}
```

The last two methods deal with players joining the game. The `onAddPlayer()` method will probably not change, unless the ruleset has to limit the number of players joining a game or the ruleset has to make sure the player can join the game at this point. Things that could be checked include if there is a limited number of races that can be played, or if there is an available planet for the player to have as a homeworld. The second method, `onPlayerAdded()`, is where the player is set up ready to play the game. This can include; creating initial designs, setting homeplanet, creating any initial resources (like fleets), and so on.

```
bool MyGameRuleset::onAddPlayer(Player* player){
  Logger::getLogger()->debug("MyGame onAddPlayer");
  return true;
}

void MyGameRuleset::onPlayerAdded(Player* player){
  Logger::getLogger()->debug("MyGame onPlayerAdded");
}
```

Note that we use the `Logger`'s `debug()` method to send the logging messages instead of `info()` like we did in `initGame()` and the other game

methods. There are five logging levels, ranging from "debug" to "error". In the normal operation of the server, debug is used for debugging messages and the next logging level up, "info", is used to inform the game administration of interesting things that are happening, such as a player logging in or the end of turn starting. There are already "info" logging messages generated by the core server when a player is created, so it's not necessary to add more.

13.1 libtool/libltdl magic

There is a little bit of trickery to allow libtool and libltdl work across multiple platforms. When the tpserver-cpp loads a module, it locates the `tp_init()` function, that has to be defined in an `extern ''C''` block. To allow multiple modules to be linked in at link time for platforms that don't support dynamic loading, each module must have a different prefix added to the `tp_init` name to make them unique.

The `tp_init()` function itself is fairly straight forward. For ruleset modules, it just sets the ruleset in `Game` to a new instance of the Ruleset class, returning the return value from the `setRuleset()` call.

```
extern "C" {
  #define tp_init libmygame_LTX_tp_init
  bool tp_init(){
    return Game::getGame()->setRuleset(new MyGame::MyGameRuleset());
  }
}
```

13.2 The minimum ruleset

Now we have a minimal ruleset that loads, initialises, and generally does nothing. When loaded and started, you should see logging messages like the following.

```
tpserver-cpp> game ruleset mygame
2007-05-29 01:01:59 < Info  > Loaded plugin "libmygame" sucessfully
2007-05-29 01:01:59 < Info  > Loaded game mygame sucessfully
tpserver-cpp> game load
2007-05-29 01:02:12 < Info  > Loading Game
2007-05-29 01:02:12 < Info  > MyGame initialised
2007-05-29 01:02:12 < Info  > Creating Game
2007-05-29 01:02:12 < Info  > MyGame created
tpserver-cpp> game start
2007-05-29 01:02:44 < Info  > Starting Game
```

```
2007-05-29 01:02:44 < Info  > MyGame started
tpserver-cpp>
```

Now while this is a working ruleset, there isn't a lot to do if you try to play the game. So the next thing is to create some object types, and create a universe.

Chapter 14

Developing Object Types

A ruleset has to set what types of object can exist. To create these object types, we create `ObjectType` and corresponding `ObjectBehaviour` subclasses, using the various `ObjectParameter` classes to store the actual data.

As mentioned in section 10.3, the common object properties are held in the `IGObject` class, and the different types of objects have their data held in the `ObjectParameter` classes. This allows an object to change it's type—for example, from a ship object to a debris object when it's destroyed.

The `Object` class holds the data for the object in a list of `ObjectParameterGroup`s, each which contain a list of `ObjectParameter`s. There are a number of `ObjectParameter` types defined, including;

- 3D Vectors for Position, Velocity and Acceleration,
- OrderQueues,
- References,
- Reference quantity lists,
- Resource List,
- Size,
- Media URL.

When creating a new object type, the two main tasks are to set up the `ObjectParameter`s and `ObjectParamterGroup`s in a subclass of `ObjectType`, and to provide a usable interface to the ruleset in a subclass of `ObjectBehaviour`. Two parallel class hierarchies are formed, one for `ObjectType`s and one for `ObjectBehaviour`s.

14.1 Universe ObjectType

We shall now create an "universe" object type. This object encompasses all objects in the game, and is the high level container. The Universe object will have position, velocity and size, and also an 'age' field, to indicate how old the universe (and the game) is.

The first stage is to create the `ObjectType` for the Universe. We only need need three methods, including the constructor and destructors, the other method creates an object instances of the corresponding `ObjectBehaviour`. The header looks like:

```
#include <tpserver/objecttype.h>

class UniverseType : public ObjectType{
  public:
    UniverseType();
    virtual ~UniverseType();

  protected:
    ObjectBehaviour* createObjectBehaviour() const;
};
```

The server only needs one instance of the `UniverseType` class, as it is only used to create and describe Universe type objects. The destructor is empty and does nothing. The `createObjectBehaviour()` we will come back to in the next section.

Almost all of the work of the `UniverseType` is done in the constructor as it sets up a few things. These don't have to be done in the constructor, but have to be done before giving the `UniverseType` object to the `ObjectTypeManager`—in this case it is easier to do it in the constructor.

First, an `ObjectParameterGroupDesc` is created for the locational information.

```
//in the constructor of UniverseType
ObjectParameterGroupDesc* group = new ObjectParameterGroupDesc();
group->setName("Positional");
group->setDescription("Positional information");
group->addParameter(obpT_Position_3D, "Position", "The position of the object");
group->addParameter(obpT_Velocity, "Velocity", "The velocity of the object");
group->addParameter(obpT_Size, "Size", "The size of the object");
addParameterGroupDesc(group);
```

A second `ObjectParameterGroupDesc` is created for the informational data, the age in this case.

```
//in the constructor of UniverseType
ObjectParameterGroupDesc *infogroup = new ObjectParameterGroupDesc();
infogroup->setName("Informational");
infogroup->setDescription("Information about the universe");
infogroup->addParameter(obpT_Integer, "Age", "The Age of the universe");
addParameterGroupDesc(infogroup);
```

Lastly, the name and description of the object type is set.

```
//in the constructor of UniverseType
nametype = "Universe";
typedesc = "The Universe";
```

Full and completed UniverseType is given in Appendix B.

Part IV

Appendix

Appendix A

Chapter 13 Basic Ruleset Module

These files are available in *ex/emptyruleset/*.

A.1 mygame.h

```
#ifndef MYGAME_MYGAMERULESET_H
#define MYGAME_MYGAMERULESET_H
/*  MyGame ruleset, example for Ruleset-dev book
 *
 *  Copyright (C) 2008  Lee Begg and the Thousand Parsec Project
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 */

#include <tpserver/ruleset.h>
```

```
namespace MyGame{

class MyGameRuleset : public Ruleset{
 public:
  MyGameRuleset();
  virtual ~MyGameRuleset();

  std::string getName();
  std::string getVersion();
  void initGame();
  void createGame();
  void startGame();
  bool onAddPlayer(Player* player);
  void onPlayerAdded(Player* player);

};

} \\namespace MyGame

#endif
```

A.2 mygame.cpp

```
/*  MyGame ruleset, example for Ruleset-dev book
 *
 *  Copyright (C) 2008  Lee Begg and the Thousand Parsec Project
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 */
```

```
// System includes

// tpserver includes
#include <tpserver/game.h>
#include <tpserver/logging.h>
#include <tpserver/objecttypemanager.h>
#include <tpserver/ordermanager.h>
#include <tpserver/player.h>

// mygame includes

// header include
#include "mygame.h"

extern "C" {
  #define tp_init libmygame_LTX_tp_init
  bool tp_init(){
    return Game::getGame()->setRuleset(new MyGame::MyGameRuleset());
  }
}

namespace MyGame{

MyGameRuleset::MyGameRuleset(){
}

MyGameRuleset::~MyGameRuleset(){
}

std::string MyGameRuleset::getName(){
  return "MyGame";
}

std::string MyGameRuleset::getVersion(){
  return "0.1";
}

void MyGameRuleset::initGame(){
  Logger::getLogger()->info("MyGame initialised"};
}

void MyGameRuleset::createGame(){
  Logger::getLogger()->info("MyGame created"};
}
```

```
void MyGameRuleset::startGame(){
  Logger::getLogger()->info("MyGame started"};
}

bool MyGameRuleset::onAddPlayer(Player* player){
  Logger::getLogger()->debug("MyGame onAddPlayer");
  return true;
}

void MyGameRuleset::onPlayerAdded(Player* player){
  Logger::getLogger()->debug("MyGame onPlayerAdded");
}

} //namespace MyGame
```

Appendix B

Chapter 14 Universe ObjectType example

These files are available in *ex/universetype/*.

B.1 universe.h

```
#ifndef MYGAME_UNIVERSE_H
#define MYGAME_UNIVERSE_H
/*  Universe object classes, example for Ruleset-dev book
 *
 *  Copyright (C) 2008  Lee Begg and the Thousand Parsec Project
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 */

#include <tpserver/objecttype.h>
```

```
namespace MyGame{

class UniverseType : public ObjectType{
  public:
    UniverseType();
    virtual ~UniverseType();

  protected:
    ObjectBehaviour* createObjectBehaviour() const;
};

class Universe : public ObjectBehaviour{
      public:
Universe();
        virtual ~Universe();


void packExtraData(Frame * frame);

void doOnceATurn();

int getContainerType();

Vector3d getPosition() const;
        Vector3d getVelocity() const;
        uint64_t getSize() const;
        void setPosition(const Vector3d & np);
        void setVelocity(const Vector3d & nv);
        void setSize(uint64_t ns);
void setYear(int year);
int getYear();


};

} //namespace MyGame

#endif
```

B.2 universe.cpp

```
/*  Universe object, example for Ruleset-dev book
 *
```

```
 *  Copyright (C) 2008  Lee Begg and the Thousand Parsec Project
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 */

#include <tpserver/frame.h>
#include <tpserver/object.h>
#include <tpserver/objectparametergroupdesc.h>
#include <tpserver/position3dobjectparam.h>
#include <tpserver/velocity3dobjectparam.h>
#include <tpserver/sizeobjectparam.h>
#include <tpserver/integerobjectparam.h>

#include "universe.h"

namespace MyGame{

UniverseType::UniverseType() : ObjectType(){
  ObjectParameterGroupDesc* group = new ObjectParameterGroupDesc();
  group->setName("Positional");
  group->setDescription("Positional information");
  group->addParameter(obpT_Position_3D, "Position", "The position of the object");
  group->addParameter(obpT_Velocity, "Velocity", "The velocity of the object");
  group->addParameter(obpT_Size, "Size", "The size of the object");
  addParameterGroupDesc(group);
  ObjectParameterGroupDesc *infogroup = new ObjectParameterGroupDesc();
  infogroup->setName("Informational");
  infogroup->setDescription("Information about the universe");
  infogroup->addParameter(obpT_Integer, "Year", "The Age of the universe");
  addParameterGroupDesc(infogroup);
  nametype = "Universe";
```

```
  typedesc = "The Universe";
}

UniverseType::~UniverseType(){
}

ObjectBehaviour* UniverseType::createObjectBehaviour() const{
  return new Universe();
}

Universe::Universe(){
}

Universe::~Universe(){
}


void Universe::packExtraData(Frame * frame){
  frame->packInt(((IntegerObjectParam*)(obj->getParameter(2,1)))->getValue());
}

void Universe::doOnceATurn(){
  ((IntegerObjectParam*)(obj->getParameter(2,1)))->setValue(((IntegerObjectParam*)(o
  obj->touchModTime();
}

int Universe::getContainerType(){
  return 1;
}

Vector3d Universe::getPosition() const{
  return ((Position3dObjectParam*)(obj->getParameter(1,1)))->getPosition();
}

Vector3d Universe::getVelocity() const{
  return ((Velocity3dObjectParam*)(obj->getParameter(1,2)))->getVelocity();
}

uint64_t Universe::getSize() const{
  return ((SizeObjectParam*)(obj->getParameter(1,3)))->getSize();
}

void Universe::setPosition(const Vector3d & np){
  ((Position3dObjectParam*)(obj->getParameter(1,1)))->setPosition(np);
```

```
  obj->touchModTime();
}

void Universe::setVelocity(const Vector3d & nv){
  ((Velocity3dObjectParam*)(obj->getParameter(1,2)))->setVelocity(nv);
  obj->touchModTime();
}

void Universe::setSize(uint64_t ns){
  ((SizeObjectParam*)(obj->getParameter(1,3)))->setSize(ns);
  obj->touchModTime();
}


void Universe::setYear(int nyear){
  ((IntegerObjectParam*)(obj->getParameter(2,1)))->setValue(nyear);
  obj->touchModTime();
}

int Universe::getYear(){
  return ((IntegerObjectParam*)(obj->getParameter(2,1)))->getValue();
}

} //namespace MyGame
```