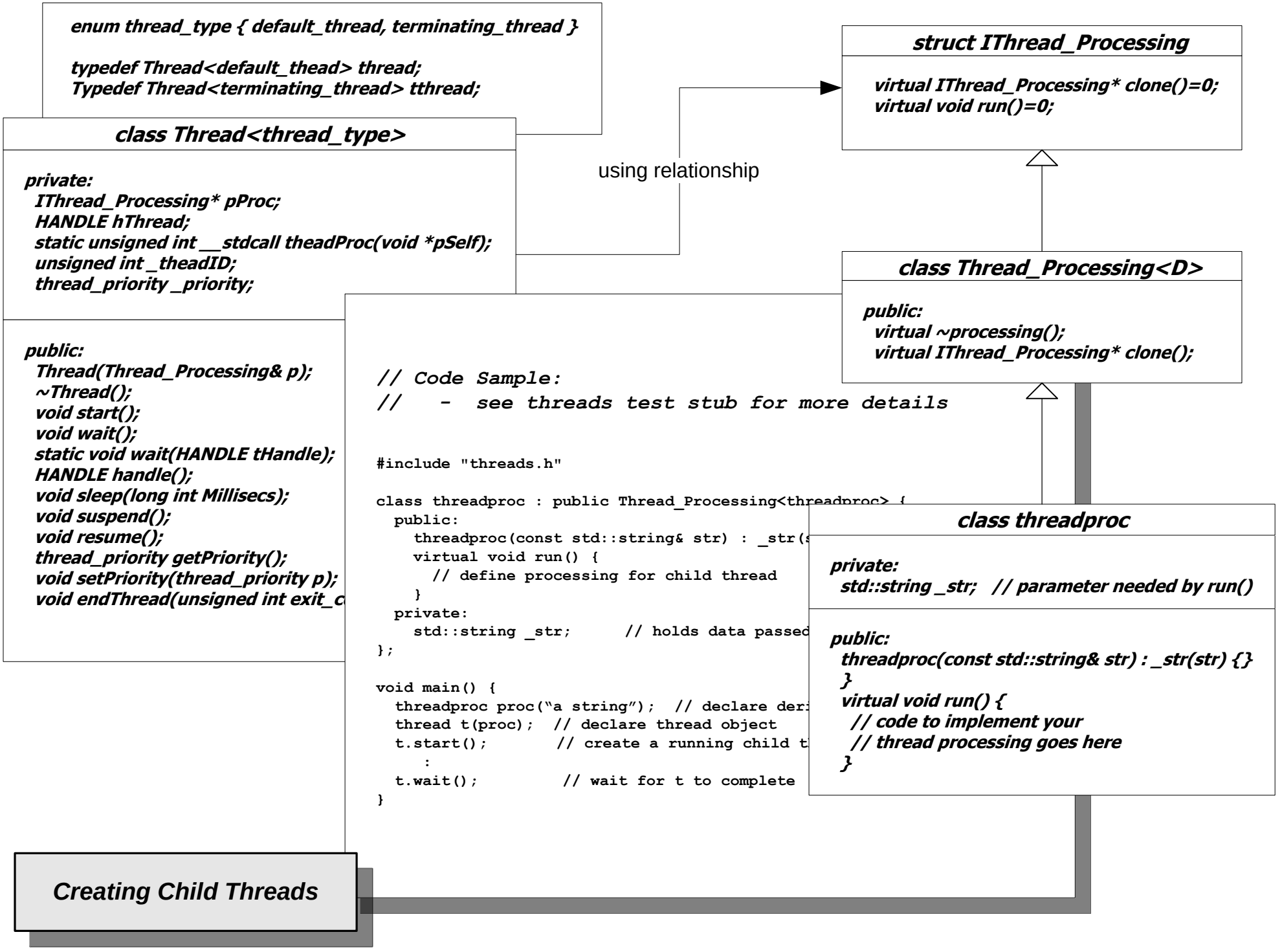
enum thread_type { default_thread, terminating_thread }
typedef Thread<default_thead> thread;
Typedef Thread<terminating_thread> tthread;
struct IThread_Processing
virtual IThread_Processing* clone()=0;
virtual void run()=0;
using relationship
class Thread<thread_type>
private:
IThread_Processing* pProc;
HANDLE hThread;
static unsigned int __stdcall theadProc(void *pSelf);
unsigned int _theadID;
thread_priority _priority;
public:
Thread(Thread_Processing& p);
~Thread();
void start();
void wait();
static void wait(HANDLE tHandle);
HANDLE handle();
void sleep(long int Millisecs);
void suspend();
void resume();
thread_priority getPriority();
void setPriority(thread_priority p);
void endThread(unsigned int exit_c
class Thread_Processing<D>
public:
virtual ~processing();
virtual IThread_Processing* clone();
// Code Sample:
// - see threads test stub for more details
#include "threads.h"
class threadproc : public Thread_Processing<threadproc> {
public:
threadproc(const std::string& str) : _str(s
virtual void run() {
// define processing for child thread
}
private:
std::string _str; // holds data passed
};
void main() {
threadproc proc("a string"); // declare deri
thread t(proc); // declare thread object
t.start(); // create a running child t
:
t.wait(); // wait for t to complete
}
class threadproc
private:
std::string _str; // parameter needed by run()
public:
threadproc(const std::string& str) : _str(str) {}
}
virtual void run() {
// code to implement your
// thread processing goes here
}
Creating Child Threads