COM S 309

Design Patterns

Case Study

THE PROBLEM CONTEXT

Context

- Company develops java virtual machine. This JVM is special in that it has a unique way to handle real-time way to handle garbage collection.

- Current code consists of **hundreds** of C files.
- Compiling current code takes at least an hour.
- Testing all the code takes several hours.

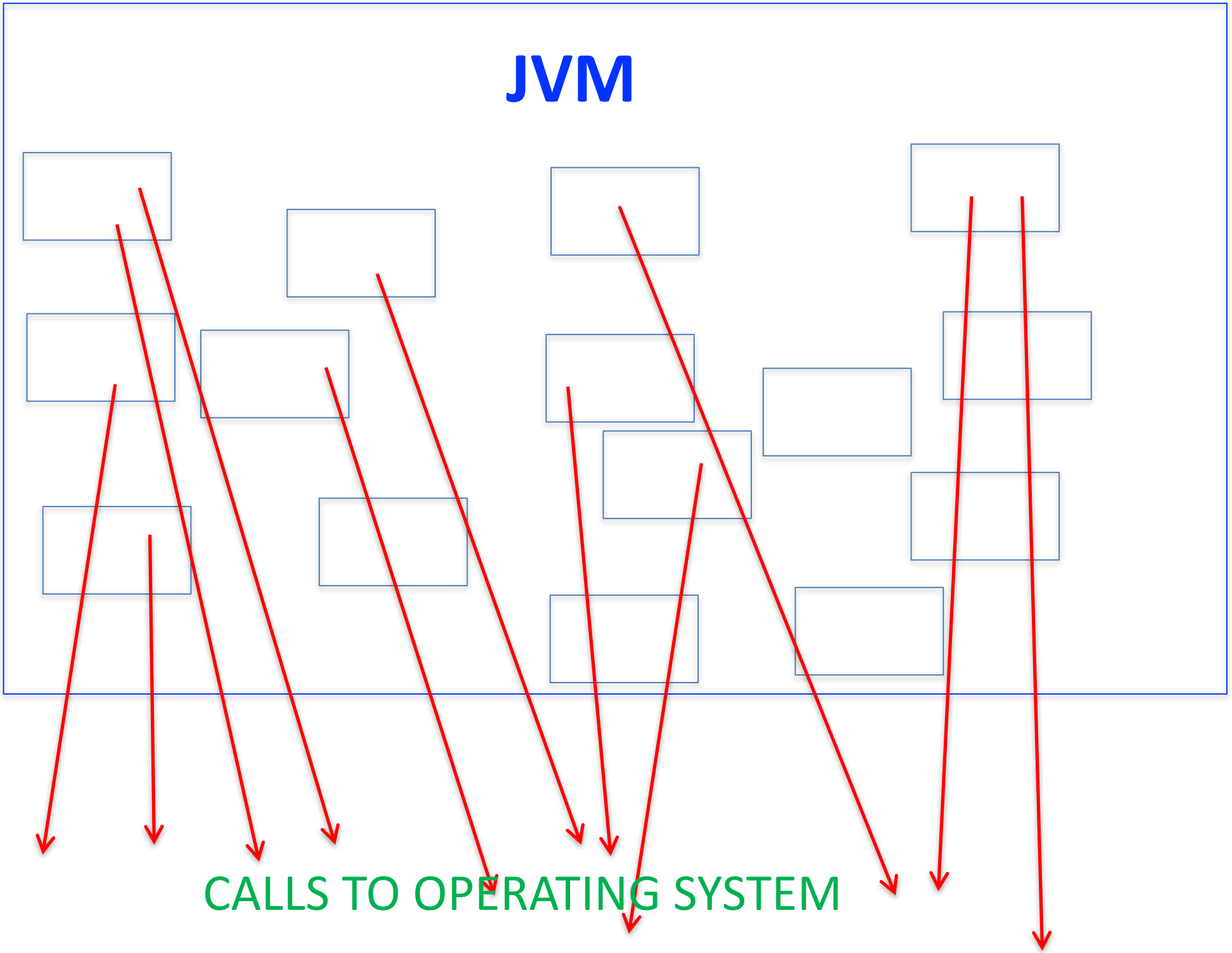
- Current code works for linux operating system.

Context

- A big customer wants to use the jvm on a windows platform!

- The hundreds of files of C has several calls to the operating systems (such as open file, read, write, create threads, allocate memory, free memory, start thread, etc). These calls are sprinkled across the many files.

- To port to Windows, each such call would need to be changed.


JVM
CALLS TO OPERATING SYSTEM

what was done.

#ifdef's were introduced in the C code

```
#ifdef LINUX
// use linux o/s call
#else ifdef WINDOWS
// use windows o/s call
#endif
```

This meant that EACH occurrence of an o/s call was modified. That's hundreds of changes in hundreds of files.

ISSUES/PROBLEMS

Issues/Problems

1. Time-consuming and error prone manual changes in hundreds of files. It took us a long time to make all the changes.
2. We had to recompile and re-test all the code several times. This again took a long time.
3. Several bugs were introduced because of subtle errors introduced during the modifications.
4. Note that simultaneous to these changes – there was other development work in progress and this major change sort of affected everybody.

More problems

- It was soon necessary to also port to another platform (pSOS operating system).
  - This lead to another round of modification to ALL the files – adding another layer of #ifdefs!
  - Much more time. Many more errors. Really tedious and painful.
- The code started looking really messy now and was harder and harder to read and understand. This added to maintainability issues.
- Porting to a NEW o/s remained really difficult, time-consuming, tedious, and error-prone.

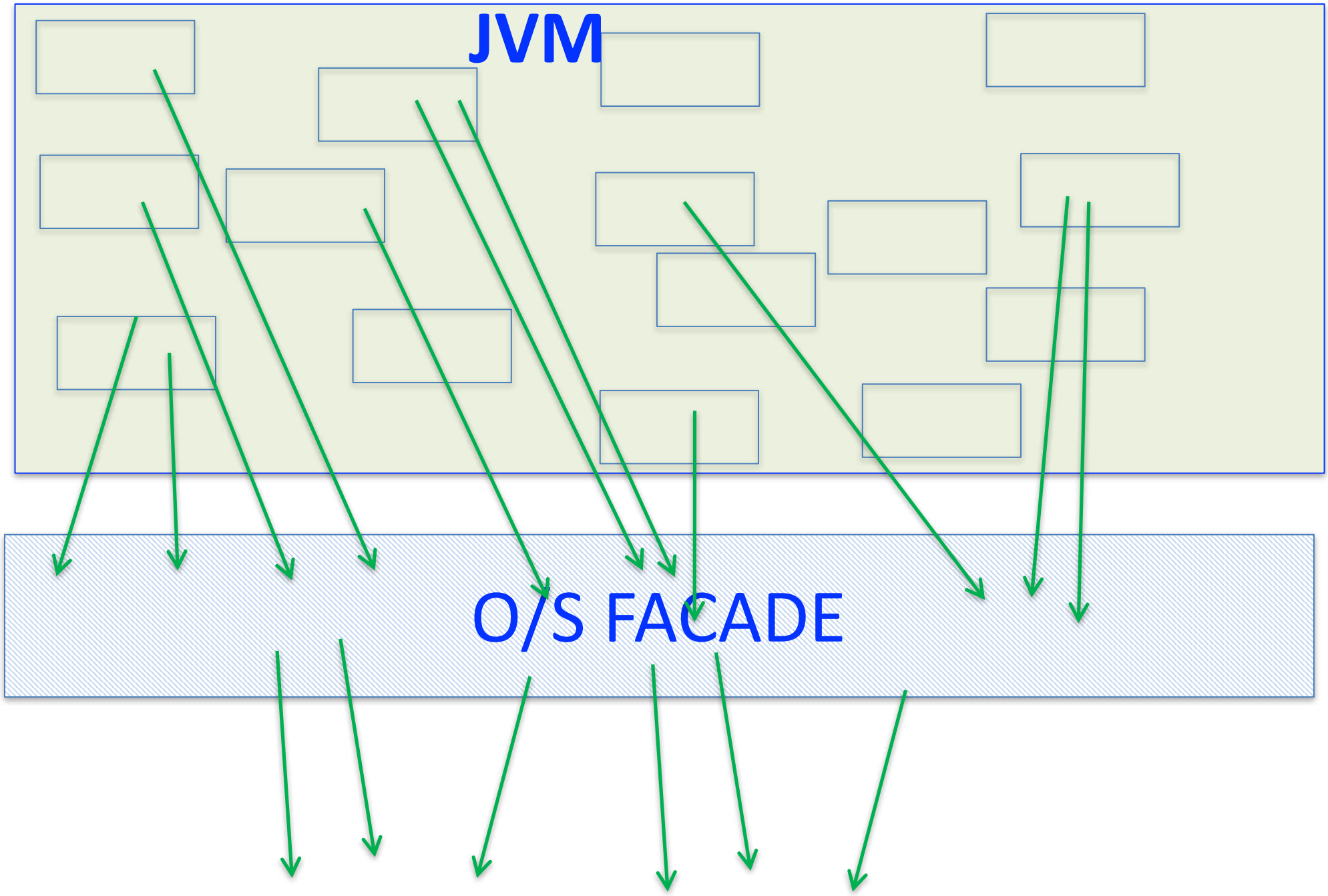
FINAL SOLUTION

Solution.

- It was decided to add a NEW layer which would consist of OUR proprietary o/s façade.

- We created an IMPLEMENTATION of this o/s façade for each different o/s.

- All of our existing code was changed to make calls to our o/s façade – getting rid of all the #ifdefs.


JVM
O/S FACADE
CALLS TO OPERATING SYSTEM

BENEFITS OF THE SOLN

Benefits

1. Got rid of all #ifdefs (extra codes) in our files. Our files were left uncluttered and easier to maintain.
2. Our files had calls to our facade. When porting to a new o/s, we did not need to make a single change in our jvm code. Thus, there was no need to re-compile and to re-test these files.
3. The developer doing the porting just had to focus on implementing all the facade code for the new o/s and to compile and to test that facade. This saved an enormous amount of time and resulted in less errors being introduced.
4. Thus, porting to a new o/s became a much simpler and mundane activity leading to major business benefits. Earlier – it was truly a nightmare proposition!

A FEW OTHER LESSONS

During development time, it is often difficult to think of how to make the code better. A standard way is to follow coding standards. But that does nothing towards making the design better!

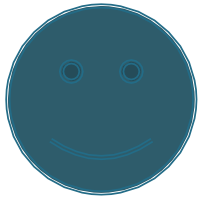
One way to making design better is to think of how will maintainers work be facilitated:

- In the case study, porting happened to be an important recurring maintenance activity. Thinking of how to make this process easier leads to a better design.
- Maintenance activities would also be things like
  - adding new screens, new tables
  - switching backend to another database.
  - switching a deprecated library to a better library

- Another way to making design better is to think about how application developers work will be facilitated.
  - Developers of new code for the jvm (i.e. application developers - where application is jvm) can start directly using the o/s facade calls and not worry about the specific o/s. This helps reduce complexity.
  - note that new porters will also not have to think of the entirety of the o/s calls. Instead, they will need to only focus on implementing the calls used in the o/s facade. This helps reduce complexity.

- In general, it helps to think about each of the roles shown on the next page.

Roles



Utility Developer

Maintainer

Application Developer

Utility Developer - **develops** classes, packages – which are used by others (for example libraries)

* Their code does not use client code or maintainer code.
* They may make calls to maintainer code (via interfaces).

Maintainer – EXTENDS, ADDS, DEBUGS, MODIFIES utility code without breaking application code.

Application Developer – **uses** classes, packages developed by utility developer – i.e. they build CLIENT code – i.e. USER code

* They just need to know how to USE the utilities. Not details of implementation.

In our case study, utility is Operating System, Maintainer builds and maintains the O/S façade and application developer is developer of the JVM code.

BENEFIT OF DESIGN PATTERNS

Benefits of design patterns

The facade pattern was shown in the case study.

There was a specific problem and the facade pattern provides a solution for that problem. Other designers can use that pattern to solve similar problems in their code.

Thus, knowledge of design patterns provide the following benefits

- Enables reuse of software design ideas.
- makes expert knowledge and design trade-offs widely available.
- helps developer-developer communication (by forming a common vocabulary) and helps in documentation and enhanced understanding.
- Eases transition to OO technology.