DESIGN PATTERNS

WHAT IS DESIGN PATTERN

- If we are dealing in cocktails I would expect you to know what is “Screwdriver”, “Bloody Mary”, “Hurricane”, “Irish eye” etc.
- When a programmer converses with another programmer there is a certain vocabulary that a programmer should know.
- Most patterns come from ”must read” books
- Here POSAv2 is Pattern Oriented Software Architecture (Doug Schmidt / ACE book) and GoF is Design Patterns by Gamma, Helm, Johnson, Vlissidis

SINGLETON

- How does Singleton differs from global variables
- What is wrong with

If Singleton * sig = NULL
sig=new Singleton.

- Singleton = Global variable (or static variable) with mutex!
- Origin GoF
- note that in singleton the mutex manages the creation not usage!

FACTORY METHOD

- Factory is a hash table that matches a name to a constructor
- Factory example is socket we define type (AF_INET, AF_UNIX, SOCK_STREAM, SOCK_DGRAM)
- Origin GoF

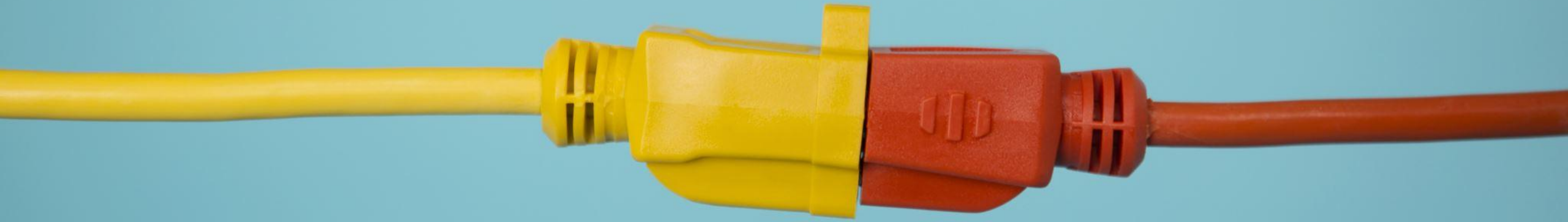
FAÇADE

- Single front (API) for multiple interfaces
- Allows changing in Plug and play method
- Example Socket Façade, File descriptor Façade
- Origin GoF

STRATEGY

- Consider using File descriptor for saving files, handling devices, randomizing numbers, sending TCP or UDP messages etc.
- All are done using read/write (or using fdopen(3) and FILE *)
- This is an example for proper use of strategy pattern.
- Origin GoF

ADAPTER



- Sometimes we have to handle interface that is not like what we are used to..
- eg. cond does not have file descriptor pattern (maybe we should use UDS)
- Origin. GoF

GUARD

- Assume that there is a code that requires only on thread to access it simultenously.
- Code may return or throw and we want to avoid deadlocks… but we can’t really tell where said code returns…
- Enter Guard.
- Mutex that is being released in the guard distractor
- Regardless of how we leave (Throw, return etc.) the critical code segment the mutex is released
- Proposed by Stroustrap in C++ Programming Langugage as “Scope Mutex”

REACTOR

- Thread that calls select(2)
- Receives function pointer to call on event
- Allows adding and removing file descriptor
- A thread that calls select(2) and calls the registered function
- Origin POSAv2 and ACE.

Proactor

Proactor is a thread that listens for incoming connections

(Possibly on multiple sockets)

When a connection occur the proactor starts new thread to handle the connection

THREAD POOL

- Several threads receives tasks (on same queue?)
- Whenever one of them is ready pop action and perform it.
- When task was performed the thread returns to the queue and waits for new tasks

Reactor/Proactor/Thread Pool

	Reactor	Thread Pool	Proactor
Number of threads	Always 1	Fixed	No max
advantages	no time wasted on context switch No time wasted on creation and termination overhead or message passing	no creation overhead. Can use multiple processors	Tasks only wait if there are system limitation. not to other tasks. Thread handling time can be very long Time not wasted on message passing
Disadvantages	Can’t use multiple processors	message passing overhead	creation overheads
When to use	handling time is short	many tasks are given so creation overhead will be significant. multiple processors can be used	Tasks can hang (for example for user input) and wait for long period of time
When not to use	When task can hang Best performance in multiple CPU environment	When tasks can hang	When many very short tasks are given (Creation overhead may be more cpu consuming than actual handling)

ACTIVE OBJECT

- Thread + Task Mission queue.
- Queue should include cond (to avoid busy waiting) and mutex
- When there is work and thread is idle pop job from queue and perform it

PIPELINE

Several Active Objects that perform works one after the other

Usages – several threads processing video to be displayed

One thread reads from file, separate audio and video.

Two threads decode (Audio and Video)

Threads for subtitles and effects

One thread corrects timing

Two threads for playback

Leader/Followers - Thread pool specialization

We are considering passing a task in a pipeline or even does to a thread poll.

Maybe we need to copy too much information

Alternative - is LF pattern

1. One leader many followers
2. Leader waits for new task.
3. When task arrives the leader
    a. quits and assign a new leader.
    b. Handles the task.
    c. when finishing to handle the task the leader join the followers

strength and weaknesses of LF pattern

Strength

- no copying of data
- no waiting in queues
- locality of data in cache

Weaknesses

- no locality of code (as in pipeline)

It's worth checking on case-by-case but my experience is that pipeline (especially if copying only a pointer) works better.

When to use :

large code (so no locality of code anyway) and much smaller data

When dealing with large data by value and not by reference

Functors

Let say we have functions and elements to call the functions on.

We can store such pairs (functions + arguments) and calculate them later (when required) so we will only do the actual calculation if required. (otherwise we can store them as function + arguments.

C++ calls this pattern functor. (function object) and it may have some usages

Activator

The environment has limited resources e.g. mobile phone.

Activator pattern similar to function object contains a task. (functor)

We create computation objects but do not call them unless resources are available.

When resources become available only than we activate the computation.