**Objective**

read in a data file “FloridaCounties.csv” which contains about 75,000 records. You will treat each record as a task to be scheduled.

**Client**

To simulate multiple clients you must preload the data from FloridaCounties.csv into a single client. The client must store the client data in a single data structure in which each record is stored as a single string variable (i.e. perhaps an array or linked list of strings). You are not allowed to preprocess the data into its constituent parts. However you will need to read the first record from the string, which for simulation purposes contains a “wait” number. This number ranges from 0 to 5 and represents the number of seconds you should wait before attempting to pass the next job to the scheduler. Note: Many jobs have wait time of 0. In this case (to simplify the problem) you must just pass the jobs consecutively with no deliberate wait time.

The client must run on its own thread throughout the entire simulation. You should not start up the simulation until the client object is loaded i.e. all records are read in and stored.

The client is responsible for passing the entire string to the scheduler. Each such request is called a ‘job’. **Web Server**

The webserver processes data for house sales in Florida counties. As this is a simulation, the tasks are deliberately “made up”.

Before discussing the scheduler, I will describe each of the tasksto be completed for EACH job. The tasks are dependent upon the data in the comma delimited housing string. The field names forattributes in this string are as follows:



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | Wait |  | 5 |  | 10 |  | eq\_site\_deductible |  | 0 |
| 2 | Statecode |  | FL |  | 11 |  | hu\_site\_deductible |  | 9979.2 |
| 3 | County |  | CLAY COUNTY |  | 12 |  | fl\_site\_deductible |  | 0 |
| 4 | eq\_site\_limit |  | 498960 |  | 13 |  | fr\_site\_deductible |  | 0 |
| 5 | hu\_site\_limit |  | 498960 |  | 14 |  | point\_latitude |  | 30.10226 |
| 6 | fl\_site\_limit |  | 498960 |  | 15 |  | point\_longitude |  | -­‐81.7118 |
| 7 | fr\_site\_limit |  | 498960 |  | 16 |  | Line |  | Residential |
| 8 | tiv\_2011 |  | 498960 |  | 17 |  | construction |  | Masonry |
| 9 | tiv\_2012 |  | 792148.9 |  | 18 |  | point\_granularity |  | 1 |

It doesn’t actually matter whether you understand the meaning of each field, because our ***tasks are deliberately*** ***rather contrived.***

In general, you will need one or more agents to perform each task type. Note: I’m using the term agent loosely

– you could think of this as a service with multiple threads, a static function, a runtime agent etc etc etc, however you want to set upthe task. You should think about concurrency issues though in designing your solution. The scheduler will be responsible for making sure each task gets completed.

**Task 1: Object Creation**

Create an object from the record. Remember that there are multiple ways to create an object. Please think about this carefully. Task 1 has the following steps:

1. Take the house string as input, parse and tokenize it (or however you want to do this)
2. Create an object that contains all of the attributes extracted from thejob string.
3. Provide getters for any fields you’ll need to access– as per requirements in other tasks.

**Task 2: Stateless Counter (pretty useless actually!)**

1. Take a house object as input.
2. Count from 0 to eq\_site\_limit
3. Count from 0 to fl\_site\_limit
4. Count from 0 to hu\_site\_limit
5. Count from 0 to fl\_site\_limit
6. Return nothing

**Task 3: Vicinity Computer**

1. Compute the distance from point 30.0000 Latitude and-­‐81.0000 Longitude to the coordinates of the house depicted in fields point\_latitude and point\_longitude (think Pythagorus).
2. Return nothing

**Task 4: Tally by Type (stateful)**

1. Simulate access to a persistent data store by use of a shared data structure. To simulate remote database accesses add **1 second of wait time** to each update.
2. Tally by type i.e. increment counter for each type i.e. Residence:Wood or Residence:Masonry (found in fields “line” and “construction”).
3. Think BASE, i.e. basicallyavailable, soft state, eventually consistent.
4. At the end of the simulation PRINT the contents of this data structure to a file or screen.

**Task 5: Register expensive houses**

1. If eq\_site\_license >= 800,000 simulate registering the house with a remote service.
2. In lieu of registration details – wait 10 seconds.

**Schedulers**

create a simple baseline first and then a more sophisticated scheduler. Tradeoffs are most likely going to fall into the performance vs. latency vs. maintainability areas.

**Scheduler # 1: Simple thread-­‐per-­‐request scheduler: baseline**

This scheduler will serve as the baseline for our study. It should be SIMPLE. You should not make any specific efforts to use design patterns etc. You may have a thread pool however which you can throttle up and down (i.e. increase and decrease the number of threads).

As each job arrives the scheduler (using one thread per request) will ensure all tasks are completed for each job.

There is one extra scheduling issue. Tasks are completed in different orders depending on the**point\_granularity**

**(PG)** field as follows:

PG=1: T1, T2, T3, T4, T5

PG=2: T1, T3, T4, T5, T2

PG=3: T1, T5, T4, T3, T2

PG=4: T1, T4, T2, T5, T3

PG=5: T1, T2, T5, T3, T4

PG=7: T1, T3, T2, T5, T4

(There are no jobs for PG=6)

This means you will have to track the status of each job. **Scheduler #2: Event Based scheduler using design patterns**

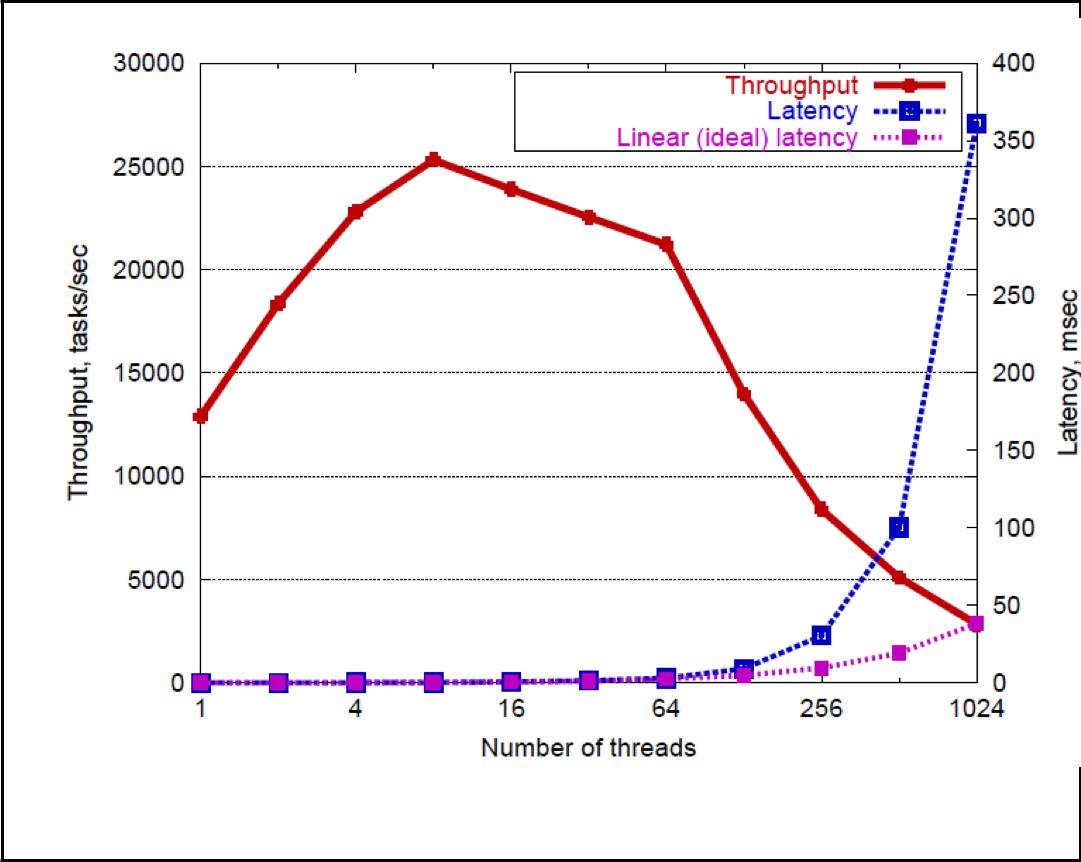
This scheduler is a more sophisticated version of the first scheduler, which does not necessarily mean it will perform better. That is for you to determine through this task.

In this part, you are required to use at least three design patterns. You may opt to use the decision trees for the *scheduler* and for *resource pooling*. You may also opt to use the *scheduler reference model* (It doesn’t have one for resource pooling though). See attached resources.

There are a number of interesting characteristics of this task– which you will need to investigate and for which you can make determinations for how best to build your scheduler.

Additionally, The architecture review board has instructed you to balance the competing needs of performance (latency and throughput) against maintainability. They realize that this scheduler is a core component of the application and is likely to evolve over time.

**Performance Analysis:**



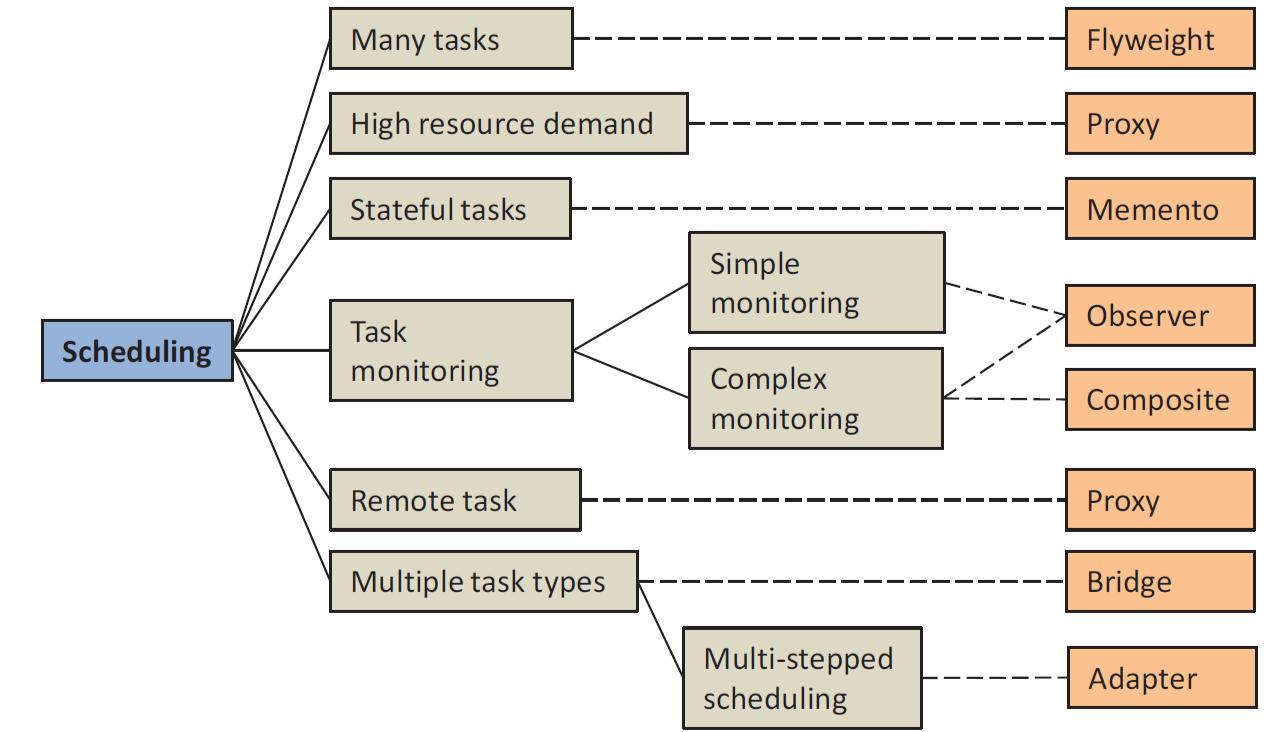
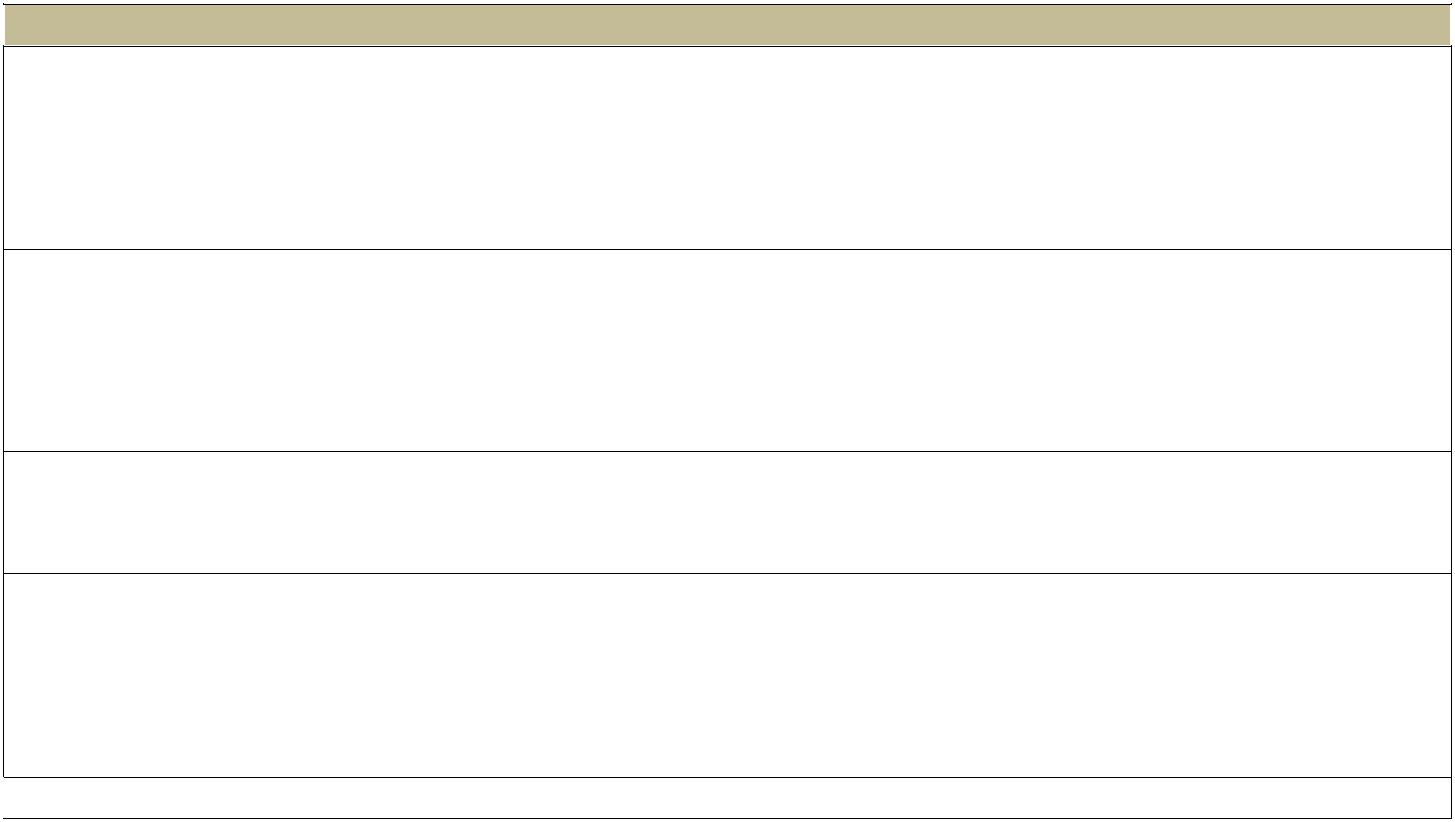
For each of your two schedulers you need to run throughput and latency tests at various sizes of threadpool. In the second event-­‐based approach you must also run tests at various batch sizes (i.e. recall the two throttles we discussed for SEDA – threadpool size and batch size). The batch size variable controls how many jobs are actively being processed by the scheduler versus just waiting in the schedulers incoming queue. You may also wish to try a dynamic approach in which these two variables are throttled automatically according to size of the incoming queue etc.

You are expected to minimally report onfixed thread pool size of 1, 10, and 20 – however you could report on more values if you wish. Results should be presented in a graph similar to the oneon the right. You will need to put on your architect’s hat and **analyze and discuss the results**.

**Maintainability:**

For each of your solutions, discuss maintainability issues. This only need to be a discussion of where you believe you have designed flexibility, extensibility into your design, and where you have chosen not to.

**Resources for scheduling and resource pool tactics**



 **Solution**  **Many tasks:** If the scheduler is responsible for scheduling a large number of tasks, incurred memorycosts may be high and it may be necessary to minimize memory usage. Tasks share intrinsic state such as task priority or task type, while also exhibiting individual properties such as start time, resources required, and so on. The **flyweight** pattern reduces the memory resource requirements, and also reduces time needed to start a task.

**High resource demand** When individual tasks have high resource demands (e.g. high memory), it is moreefficient to create them immediately prior to use and destroy them immediately afterwards. The**proxy** pattern can be used so that a proxy object serves as a stand-­‐in for the task while it waits in the queue. The scheduler is unaware that the proxy exists. However, the proxy creates, invokes, and destroys the task as soon as it is scheduled for execution.

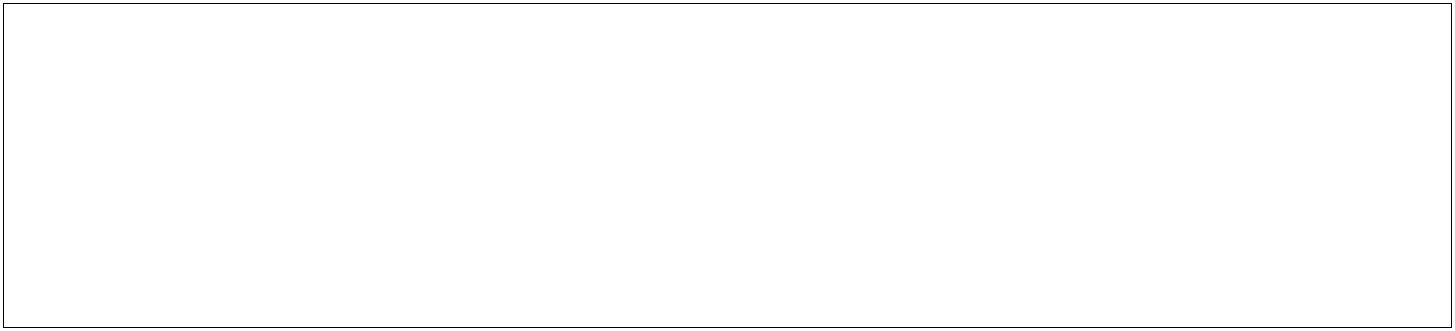
**Stateful tasks** When the state of the task needs to be preserved between scheduled runs, the**memento** pattern is used to preserve state at the end of a run, and restore previous state at the beginning of the next run.

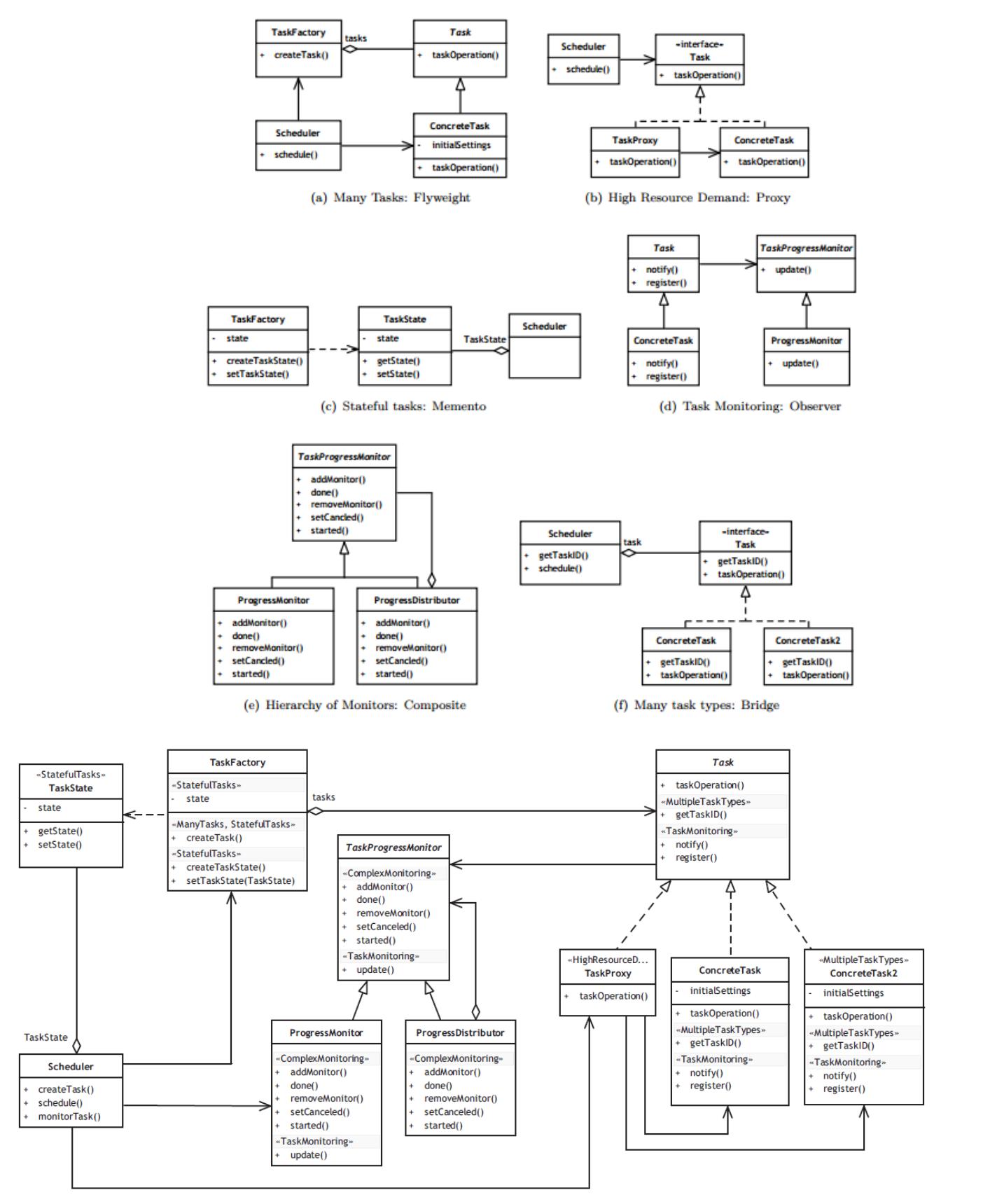
**Task monitoring** Tasks often need to be monitored to ensure that they are active and progressing. The **observer pattern** allows a task progress monitor (which could be the scheduler itself) to register as anobserver of the task and receive status update notifications. In more complex systems, it may be necessary to have hierarchies of monitors. In this case the**composite pattern** can be used to compose monitors into hierarchies.

 **Remote tasks** The **proxy** pattern can be used to provide a local object as a stand-­‐in for a task that is

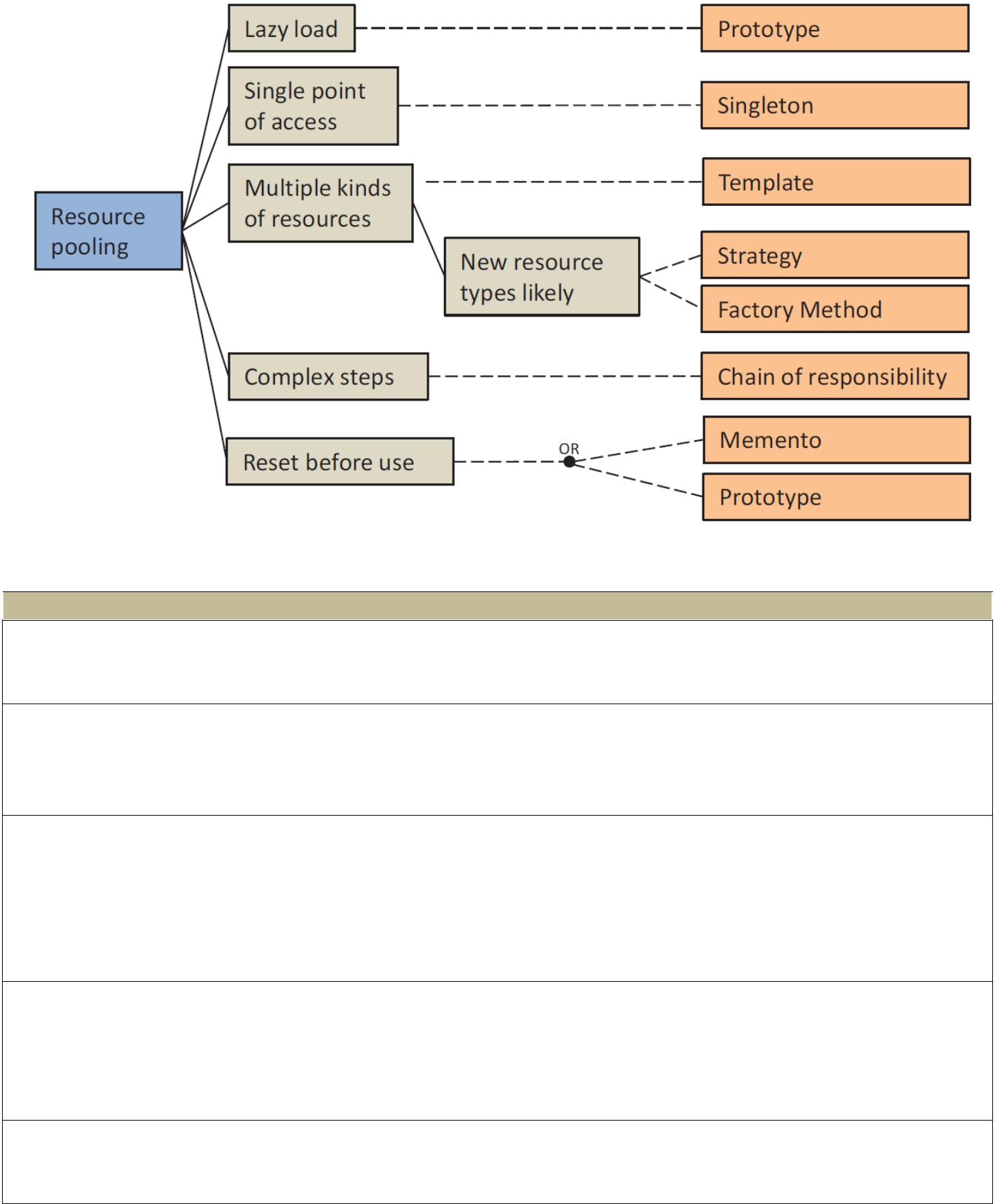


 running remotely. This is not shown in the Figure.  **Multiple task types** When a scheduler is responsible for managing multiple types of tasks i.e. MapTask,ReduceTask, DFSTask, etc, and new types of tasks may be added in the future and/or behavior of those tasks may change over time, then the **bridge** pattern can be used to create a flexible environment in which both the tasks and their behavior can change independently. When there are multiple types of tasks to be scheduled, and different types of tasks involve different steps (i.e. retrieving data, checking priorities etc) then the **adapter** pattern can be used to adapt each task with the required steps. In this case a generic schedule method is invoked for all tasks and this method is then adapted according to task type.





**A reference model for the scheduler tactic. Variability points are marked as stereo-­‐types. These stereotypes are used to reduce the model to deliver only the functionality specified.**



 **Solution**  **Lazy load** Shared resources are typically created either at start-­‐up time or upon demand. When the costof creating a resource in the standard way is high, the**prototype** pattern can be used to efficiently create a clone of a prototypical object.

**Single point of access** Access to a resource pool ismanaged by a*pooler*role. If many differentcomponents need to access the pool there can be significant overhead for establishing and managing a sharing scheme. The problem can be addressed through using**singleton** to ensure that there is only one instance of the pooler at runtime.

**Multiple kinds of resources**When there are multiple types of resources (i.e. thread pooling, connectionpooling, session pooling), and if the types are stable, i.e. it is unlikely that new types will be introduced, the **template** method can be used to customize resource management according to resource type. When new types are expected to be introduced the**factory** pattern can be used to create the pools and their associated resources (sometimes referred to as ponds), while the**strategy** pattern can be used to select the appropriate factory method according to type.

**Complex steps** When the task of creating, managing, and using a resource pool is complex, the**chain of responsibility** pattern can be used to decouple the sender of a request from its receiver. For example aseries of requests for checking the existence of the pool, checking the number of objects, or checking the maximum allowed number of objects, can be passed as a command along a chain of objects until the request is handled.

**Reset before use** A resource pool recycles resources. Before a recycled resource can be reused it needsto be reset to its original state. Use either**memento** to reset the resource to its original state, or **prototype** to create a new clone before each use.