StatisticsFurtherSpeculations

<?xml version="1.0" encoding="utf-8"?>
<html>

Logging Notes

From looking at Liam's thoughts here, I would agree that it makes sense to use the log4j system with the DAppender to manage the deposit of logs in the database. This should basically give us the pros from both the suggestions I made without any significant cons.

Further to this, then, I have been thinking about how this might be best utilised in DSpace, and think that taking Liam's LogManager suggestions further might be the best solution. That is, we would encapsulate all logging actions inside our own LogManager class, so as to insulate the user from interfacing directly with log4j and it's MDC concept. A class specification might be:

```java
public class LogManager {
    public void setParam(String key, String value) {
        if (!LogParams.validate(MDC.get("action"), key)) {
            throw new LogParameterException(); // for example
        } else {
            MDC.put(key, value);
        }
    }
    public void setAction(String action) {
        if (!LogParams.validate(action)) {
            throw new LogParameterException(); // for example
        } else {
            MDC.put("action", action);
        }
    }
}
```

So this would basically provide an interface to the MDC class, but also will provide an interface to the logging action. The purpose of this is to move knowledge of how logging actually works, and it's specific relationship to log4j away from the user (i.e. the module writer). Shamelessly stealing Liam's example, this would then implement as:

```java
LogManager log = new LogManager(rowse.class);
log.log(LogManager.INFO, logInfo);
log.clearAll();
```

Constrained actions and parameters

I am having trouble deciding if constraining parameters and actions for logging is a good idea or not. It would enforce consistent logging, at the cost of perhaps annoying the programmer, having to jump through some hoops to log an activity, and not being able to change a logging message structure without making modifications elsewhere (probably in another class). Nonetheless, here is something a bit more solid regarding this idea, should it be necessary.

The following database table is the trivial method of maintaining a link between allowed actions and parameters

```java
{ log_properties--------------action -- the name of the allowed action
   property -- the name of the allowed property for that action
}
```

So we just hold a basic map of actions to properties which we can validate against. I imagine the following LogParams class mediating with this table in various ways

```java
public static boolean validate(String action, String param) {
    // logic
    return true;
}
```

So during a logging process the following calls to this class might be made:

```java
 LogManager log = new LogManager(Loader.class);
 log.log(LogManager.INFO, logInfo);
```

I've also put a register() method in there so that modules could provide XML files containing their logging actions and parameters, which could be read in at build time, and pushed into the database through this class.

The point of this, as I see it, is that we get to enforce consistent logging standards for each logging action, and also have a registry of actions and parameters that will allow for the building of reports more easily. For example, we could have a search interface on the reports section that allows you to select an action, and make a search based on one of the parameters. With a registry of actions and parameters this would be a straightforward exercise. Also, it may be useful for building tables of results showing actions and selected parameters, and combining different logging events with each other through like parameters.

Does this section go to far?
I don't know if this is a step too far, but here it is for consideration anyway.

As DSpace becomes more modular, it may be sensible to abstract this logging process even further, so that modules can implement their own customised logging features. I briefly brought this up in my last post on this, but here it is a little more formalised. Basically I am wondering whether we should encourage each module to extend the LogManager class to create a ModuleLogger (e.g. TapirLogger). The idea would be to have this do all the leg work with regard to logging on a module by module basis, and to keep logging code clear from the main logic. For example:

```java
// Wrapping the actual "log" call in the LogManager means that log4j would not be able to record line numbers (they would all be line number of code in the LogManager class....). But that's probably not important in the whole scheme of things - I don't think we log the line nos now anyway.... and I think reducing the pollution of functional code by logging gubbins is probably worth this drawback.

// In response to the above I have an number of comments/ideas....

// Encourage each module to extend the LogManager class to create a ModuleLogger (e.g. TapirLogger). The idea would be to have this do all the leg work for the logging features. I briefly brought this up in my last post on this, but here it is a little more formalised.

// Actually it seems clear that what we need is to be able to create a class for logging that is separate from the main business logic of the modules.

// Speculations this time from Liam ==

// Question is, is this just too much abstraction, to the point where it's just a pain to use? Answers on the back of a Wiki ...

// Yet Further Statistics & Logging

// This combines actions of both directly setting parameters, and passing them wholesale to the module specific log manager.

// In response to the above I have a number of comments/ideas....

// Wrapping the actual "log" call in the LogManager means that log4j would not be able to record line numbers (they would all be line number of code in the LogManager class....). But that's probably not important in the whole scheme of things - I don't think we log the line nos now anyway.... and I think reducing the pollution of functional code by logging gubbins is probably worth this drawback.

// I kind of liked the idea of the subclasses of LogManager - but after some consideration I think it's creating too much of a burden on the module programmer. I think adding log actions should be just a config change, that install-configs should take care of.

// We should take steps to make logging as convienient as possible. Asically we want the logging an action to be a single line to be possible - though we also want people to be able to gather params as they go if they want. For single line I suggest -

```
Where logAction() will validate first, and clear the action and all action params but not contextual params after it's done. And one should be able to use both techniques at the same time. So this way the validation "signature" (i.e. which params are required) is in the config/db and not duplicated in a java method signature, but we still have more or less the same convenience factor. I think we need to distinguish easily between action-related parameters and ones that are really context-related (thread level) and allow people to put contextual ones in easily, at any time, as it may be useful. So a separate -

```java
private static LogManager log = LogManager.getInstance(MyClass.class);
```

would be good, and anything added like this should not be flushed. One way would be for the LogManager to have a Stack containing which logging action params have been added, and pop everything off it, clearing them in the MDC as it goes, when it's done logging the action (so leaving everything else).

We might also want to facilitate addhoc logging for debugging purposes with a log.debug() method that logs at debug level and doesn't have an "action" property (but records any contextual props plus any action params already set...,)

The allowed actions and their params should probably be stored in memory while the app runs and not require a database trip - e.g. Map of LogAction objects each containing a list of params, read from the D at app initialisation (i.e. this is the data of LogParams class - though might be better to call it LogActionRegistry or something). We could validate the MDC directly, just prior to a log call - i.e. pull the action value, and check that all required params are there.

E.g. LogActionRegistry.validate(MDC)

Just to clarify how the LogManager class would work, given all of that. The client class instantiates a static reference to an instance of the LogManager e.g.:  

```java
private static LogManager log = LogManager.getInstance(MyClass.class);
```

Each instance of LogManager needs a reference to the log4j logger for the class it's logging for -

```java
private static LogManager log = LogManager.getInstance(MyClass.class);
```

We might also want to facilitate addhoc logging for debugging purposes with a log.debug() method that logs at debug level and doesn't have an "action" property (but records any contextual props plus any action params already set...,)

The allowed actions and their params should probably be stored in memory while the app runs and not require a database trip - e.g. Map of LogAction objects each containing a list of params, read from the D at app initialisation (i.e. this is the data of LogParams class - though might be better to call it LogActionRegistry or something). We could validate the MDC directly, just prior to a log call - i.e. pull the action value, and check that all required params are there.

E.g. LogActionRegistry.validate(MDC)

Just to clarify how the LogManager class would work, given all of that. The client class instantiates a static reference to an instance of the LogManager e.g.:  

```java
private static LogManager log = LogManager.getInstance(MyClass.class);
```

Each instance of LogManager needs a reference to the log4j logger for the class it's logging for -

```java
private static LogManager log = LogManager.getInstance(MyClass.class);
```

```java
Unknown macro: {private Logger logger4j;logger4j = Logger.getLogger(inClass);}
```