## 220 2012/13

1a Consider Rainsberger’s four elements of simple design

i) List the four elements, in order

Correct behaviour, Minimal duplication, Maximum clarity, Fewer elements

ii) Pick two of these elements and explain briefly why they are important

Correct behaviour – a design that is brilliant and elegant but doesn’t do what we need it to do is useless in practice where companies / clients need products to accomplish given tasks.

Minimal duplication – reducing the number of times particular pieces of code are duplicated in the code base is important, since any change we make to one of these duplicates needs to be done in all the other instances. This is error-prone and wastes developers’ time.

1b Look at the code in Listing 1 on the next page. Show – by sketching code and diagrams – two different ways that you could remove the duplication from this code by applying two different design patterns.

i)

Template method pattern

abstract class Alert {

abstract void sendAlert(String message);

private static final double THRESHOLD = 10.0;

private final MonitoringSystem monitoringSystem;

Alert(MonitoringSystem monitoringSystem) { ... }

public void run() {

// ...

sendAlert(message);

// ...

}

}

class EmailAlert extends Alert {

public EmailAlert(MonitoringSystem monitoringSystem) { ... }

@:Override void sendAlert(String message) { ... }

}

class SmsAlert extends Alert {

public SmsAlert(MonitoringSystem monitoringSystem) { ... }

@:Override void sendAlert(String message) { ... }

}

ii)

Strategy pattern

class Alert {

private static final double THRESHOLD = 10.0;

private final MonitoringSystem monitoringSystem;

private final AlertStrategy alertStrategy;

Alert(MonitoringSystem monitoringSystem, AlertStrategy alertStrategy) { ... }

public void run() {

// ...

alertStrategy.alert(message);

// ...

}

}

interface AlertStrategy {

void alert(String message);

}

class EmailAlertStrategy implements AlertStrategy {

void sendAlert(String message) { ... }

}

class SmsAlertStrategy implements AlertStrategy {

void sendAlert(String message) { ... }

}

1c Considering your two solutions from part b) comment on which solution is preferable and why.

Strategy pattern seems more appropriate here. It may be the case that there would be a different alert condition (i.e. not based on sensors as in the current Alert classes) – then we could re-use our AlertStrategies very easily.

2a Describe briefly what a *future* is.

A future is an object that represents the result of an asynchronous query / function call, and provides a way to check if this result is available yet, as well as a way to synchronously wait until it is.

For the remainder of this question, consider that we want to calculate the mean size (in bytes) of web pages on the Internet. We want to write a function of the following form:

double meanSizeInBytes(List<URL> urls) { ... }

We have access to a UrlFetcher, which we can use like this:

Fetcher fetcher = new UrlFetcher();

byte[] content = fetcher.fetch(url);

b Show how this computation can be expressed as a map-reduction.

map: maps a URL to its content length

int map(URL url) return fetcher.fetch(url).length;

reduce: sums two content lengths

int reduce(int len1, int len2) return len1 + len2;

urls.map(\x → fetcher.fetch(url).length / n).reduce(\x1, x2 → x1 + x2)

c Show – by sketching code and possibly diagrams – how you could implement the map phase in parallel on one machine. Do not use new Thread().

We could use a executor pool that would consume our list of URLs and run mapping functions in separate threads whenever they are free.

As of Java 8 you can now use: urls.parallelstream().map etc (this came out after the paper)

d Later we adapt the system to perform different analyses on the web pages. The fetcher is the performance bottleneck – fetching each page is quite slow. Describe how you could apply another design pattern to speed up the use of this component and make the system more responsive.

We could use a caching proxy pattern – then we only fetch each page once, and return a stored copy from memory on subsequent requests, allowing us to do multiple different analyses on the same set of URLs without requesting data more than once.