Problem Set 2: Composing Concepts

Concept Questions

1. Contexts
Contexts are used to separate the generation of nonces for different situations. This makes sure that the same short string can potentially be used in different contexts without causing a conflict.
In the URL shortening application, the context would be the `shortUrlBase`. For example, `example.com` and `mydomain.com` can each have their own set of generated short strings. So, a nonce only needs to be unique within its context, not necessarily across all contexts.

2. Storing Used Strings
NonceGeneration stores a set of used strings because it needs to remember which strings have already been generated. This allows the concept to guarantee that a newly generated nonce has not been used before in the same context.
If the implementation uses a counter instead, the counter represents how many nonces have already been generated for a context.
For example:
Counter = 0 → no strings have been generated.
Counter = 1 → one string has been generated.
Counter = 2 → two strings have been generated.
The abstraction function maps the counter to the corresponding set of strings generated by that counter.
For example, if the generated strings are `"1"`, `"2"`, and `"3"`, then:
text
counter = 3
represents:
text
used = {"1", "2", "3"}
So, the counter is an implementation of the abstract `used` set.

3. Words as Nonces
One advantage of using dictionary words is that the resulting short URLs are easier for users to remember.
For example:
text
example.com/coffee
is easier to remember than:
text
example.com/X7kP29
One disadvantage is that the number of available dictionary words is limited. This could make it harder to generate unique short URLs when many users are using the service.
To modify `NonceGeneration`, the concept could store a set of available dictionary words and a set of words that have already been used in each context.

text
concept NonceGeneration [Context]
purpose generate memorable unique words within a context
principle each generate returns a word not returned before for that context
state
  a set of Contexts with
    a used set of Strings
actions
  generate (context: Context): (nonce: String)
    effect returns an unused dictionary word and adds it to the context's used set

Synchronization Questions

1. Partial Matching
The first sync only needs `shortUrlBase` because its purpose is to generate a nonce for the given domain.
The `targetUrl` is not needed to generate the nonce.
The second sync needs both `targetUrl` and `shortUrlBase` because after the nonce is generated, the system needs all three pieces of information to register the shortening:
the generated nonce
the short URL base
the target URL
Therefore, the first sync is only concerned with generating a nonce, while the second sync is concerned with creating the actual URL shortening.

2. Omitting Names
The convention is not used in every case because sometimes the argument name and the variable name are different.
For example:
text
context: shortUrlBase
Here, `context` is the argument name and `shortUrlBase` is the variable being passed to it.
If the names were omitted, it would not be clear which variable is being assigned to which argument.
Therefore, names can only be safely omitted when the argument name and variable name are the same.

3. Inclusion of Request
The request action is included in the first two syncs because those actions are initially triggered by a user's request to shorten a URL.
The third sync does not need a `Request` because it is triggered by the completion of another concept's action.
Specifically:
text
UrlShortening.register()
causes:

text
ExpiringResource.setExpiry()
So the third sync is an internal synchronization between concepts rather than a direct response to a user request.

4. Fixed Domain
If the application only used a fixed domain such as `bit.ly`, the request would no longer need to provide `shortUrlBase`.
The synchronizations could be changed to use the fixed domain directly.
text
sync generate
when Request.shortenUrl (targetUrl)
then NonceGeneration.generate (context: "bit.ly")
Then the register synchronization could be:
text
sync register
when
  Request.shortenUrl (targetUrl)
  NonceGeneration.generate (): (nonce)
then UrlShortening.register (
  shortUrlSuffix: nonce,
  shortUrlBase: "bit.ly",
  targetUrl
)
This works because the domain is now fixed instead of being provided by the user.

5. Adding a Sync
When a short URL expires, it should also be deleted from `UrlShortening`.
text
sync deleteExpired
when ExpiringResource.expireResource (): (resource)
then UrlShortening.delete (shortUrl: resource)
The `resource` returned by `expireResource` represents the expired short URL. That value is then passed to `UrlShortening.delete`.

Extending the Design: Analytics

1. Additional Concepts
Two additional concepts can be used to support analytics: `Analytics` and `ShorteningOwnership`.
Analytics
text
concept Analytics [ShortUrl, User]
purpose count how many times each short URL is accessed
principle after a short URL is accessed, its access count increases
state

a set of AnalyticsRecords with
a shortUrl ShortUrl
a creator User
a count Number
actions
create (shortUrl: ShortUrl, creator: User)
recordAccess (shortUrl: ShortUrl)
getCount (shortUrl: ShortUrl, user: User): (count: Number)
requires user is the creator of the short URL
effect returns the number of accesses for the short URLShorteningOwnership
text
concept ShorteningOwnership [ShortUrl, User]
purpose associate each short URL with the user who created it
principle after a user creates a short URL, that user is its owner
state
a set of Ownerships with
a shortUrl ShortUrl
a user User
actions
register (shortUrl: ShortUrl, user: User)
getOwner (shortUrl: ShortUrl): (user: User)
The `Analytics` concept stores the access count, while `ShorteningOwnership` associates each short URL with its creator.
This allows analytics to remain private to the user who registered the shortening.

2. Three Essential Synchronizations
A. When Shortenings Are Created
When a shortening is successfully registered, an analytics record is created for it.
text
sync createAnalytics
when UrlShortening.register (): (shortUrl)
then Analytics.create (shortUrl)
The ownership information can also be registered at the same time:
text
sync registerOwnership
when
UrlShortening.register (): (shortUrl)
Request.shortenUrl (targetUrl, shortUrlBase, user)
then ShorteningOwnership.register (shortUrl, user)
B. When Shortenings Are Translated to Targets
When `UrlShortening.lookup` completes, the analytics count should increase.
text
sync recordAccess
when UrlShortening.lookup (shortUrl): (targetUrl)
then Analytics.recordAccess (shortUrl)

This means every successful lookup is counted as one access.
C. When a User Examines Analytics
The user can request the number of accesses for a particular short URL.
text
sync viewAnalytics
when Request.viewAnalytics (shortUrl, user)
then Analytics.getCount (shortUrl, user)
The `user` is passed to the analytics concept so that the analytics result is only available to the creator of the shortening.

3. Additional Feature Requests
A. Allowing Users to Choose Their Own Short URLs
This feature can be implemented by adding an action to `UrlShortening` that allows the user to provide the desired short URL suffix.
For example:
text
chooseShortUrl (shortUrlSuffix, shortUrlBase, targetUrl)
The action would require that the chosen short URL does not already exist.
This feature can work with the existing analytics design because analytics only needs the resulting `shortUrl`.
B. Using the "Word as Nonce" Strategy
This feature can be implemented by modifying `NonceGeneration` so that it generates dictionary words instead of random strings.
The existing synchronizations do not need to change because they only depend on the nonce returned by `NonceGeneration`.
Therefore, this is a good example of modularity because the nonce-generation strategy can change without requiring changes to the other concepts.
C. Including the Target URL in Analytics
This can be implemented by adding the target URL to the analytics record.
For example:
text
AnalyticsRecord
  shortUrl ShortUrl
  targetUrl String
  creator User
  count Number
When a shortening is created, the target URL can be stored together with the analytics information.
This allows analytics to group different short URLs that point to the same target URL.
D. Generate Short URLs That Are Not Easily Guessed
This feature can be implemented by changing `NonceGeneration` to generate random strings that are difficult to predict.
For example:
text
X7kP29

q9Lm42
T4xZ81
Instead of sequential values:
text
abc1
abc2
abc3
The synchronization does not need to change because it only uses the nonce returned by `NonceGeneration`.
This shows the benefit of separating nonce generation into its own concept.
E. Supporting Reporting of Analytics to Creators of Short URLs Who Have Not Registered as Users
This feature would require additional support because the current design associates analytics with a registered `User`.
One possible solution is to create another concept that provides an access key for creators who are not registered users.
text
concept AnalyticsAccess
purpose allow non-registered creators to access analytics
state
  a set of AccessRecords with
    shortUrl ShortUrl
    accessKey String
actions
  createAccess (shortUrl: ShortUrl): (accessKey: String)
  viewAnalytics (shortUrl: ShortUrl, accessKey: String)
    requires accessKey is valid for shortUrl
    effect returns analytics for shortUrl
However, this adds more complexity because the system needs another way to authenticate the creator.
Therefore, I would only include this feature if supporting non-registered creators is an actual requirement. Otherwise, it may be better to keep analytics restricted to registered users.