* Semantic Hashing: embed entities (data, schema: information, behavior: knowledge) contexts and functional Relationships, Inferences and Protocol, Gestures, Actions encoded into Augmented Ontology identifiers.
* Semantic Hashing: W3C DIDs (Distributed Identifiers) of Models CRUD and Purpose behavior instances: DLT, Events Sourcing.
* Semantic Hashing: Declarative Services (OSGI) / Contexts Roles Interactions (Model hash resolver) Case Matching. OGM DOM / DCI Augmented Models Services. Facades.
* Semantic Hashing: Declarative Purpose (Context) / Gestures (Actions) / Steps (prompts / dialog) : Flow Protocol Abstractions. OGM DOM MVC DCI CDI DDD.
* Components, Layers: Stacks Contexts CDI: hashing resolution. Applications Domains: integration / interoperation / matching.
* Semantic Hashing: Incorporate Domains: What a given Application Domain is about to another Application Domain (a is Customer in ERP, is Employee in Payroll). Discovery: Models, Services. Employee with good performance has discounts as a Customer.
* Semantic Hashing: Layers / Stacks encoding. Models /Domains / Protocols. MVC / DCI HATEOAS Uniform ResourceURN encoded Quads REST Facades.
* Semantic Hashing: Layers, Models. Encodings: Representations, Protocols, Facades.
* Type Inference
  + HKT: Monads (Kinds), Transforms (Functions) from instance data.
* Augmentations pipelines: Models Monads Layers plus Service APIs.
* DDD. Augmentation: Input / Output Quad Monad.
  + Service APIs: Resources Events Dataflow.
  + Model: Plain RDF. CSPOs (Layers Representation Encodings).
  + Available Quads I/O Contexts. (case matching / hashing injections).
* DCI. Augmentation: Aggregation Monad. Contexts.
  + Service APIs: Aggregation. Registry. Contexts.
  + Model: Sets. Resources, Kinds, Contexts.
  + Available Contexts Interactions
* MVC. Augmentation: Alignment Monad. Interactions.
  + Service APIs: Alignment. Index. Interactions.
  + Model: Nested Key Value.
  + Available Interactions (roles) Data (players).
* DOM. Augmentation: Activation Monad. Data.
  + Service APIs: Activation. Naming. Data (players).
  + Model: OGM / DOM Objects. Objects / Graph Model.
  + Available Data Contexts
* CDI. Augmentation: Injection. Map Monad.
  + Service APIs: MapReduce.
  + Model: FCA Contexts Lattices. ResourceURNs.
  + URN bitstring Encoding. Injections: Hash matchings.
* DataFlow: Input / Output:
* DDD REST Endpoint. Message (Events) Interaction Layers Representations I/O DataFlows.
* Augmentations Stack: Nested / Recursive: (DDD (DCI (MVC (DOM (CDI (DOM… (RDF4J Sail layers).
* Dispatch: CQRS via type / object / command / predicate / context (CDI) case matching. Protocol: bidi dialog prompts.
* Augmentations Layers:
* Layers Models Builders: Templates / Transforms. Event driven (SAX like) parsers to / from layers models. Uniform Representations across layers, Layers specific Object Models.
* Beans Serialization. Templates: case matching. URN bitstring Encoding. Injections: CDI Type / HKTs / instances / functional contexts / predicates hash matchings.
* Pipelines: reactive streams runtime. Layers bundles (RDF4J Sails, Reactive-X / OSGi / Vert.x) DataFlows. DDD RDF I/O Augmented request (augments models : CDI) / response (augments replies : CDI) "smart" CQRS RESTFul HATEOAS Interfaces.
* DDD. Input / Output:
  + Input: RESTful HATEOAS Representations (Events / Messages)
  + Process input (Input Model)
  + Augmentation: I/O Monad.
  + Process output (Output Model)
  + Output: RESTful HATEOAS Representations (Events / Messages)
  + Available Quads I/O Contexts. Process input quads and populate Quads Contexts set layout.
* DCI / ESB. Use Cases. Contexts. Controllers. Registry:
  + Input
  + Process input (Sets)
  + Augmentation: Aggregation Monad.
  + Process output (Sets)
  + Output
  + Purpose
  + Role
  + Actor
  + Context
  + Gestures
  + Actions
  + Available Contexts Interactions: CDI Contexts. Process input Sets and populate Context Views.
* MVC. View. Interactions. Index:
  + Input
  + Process input (Nested Key Value)
  + Augmentation: Alignment Monad.
  + Process output (Nested Key Value)
  + Output
  + Service APIs: Alignment. Index. Interactions.
  + Model: Nested Key Value.
  + Alignment. Index / Interactions (roles).
  + Available Interactions (roles) Data (players). Process views keys and instantiate OGM DOM.
* DOM. Models. Data:
  + Input
  + Process input (OGM DOM)
  + Augmentation: Activation Monad.
  + Process output (OGM DOM)
  + Output
  + Model: OGM / DOM.
  + Activation. Naming / Data (players). Available Data (players) Contexts. Activate OGM DOM Objects and populate CDI pool for dependency resolution.
* CDI:
  + Input
  + Process input (ResourceURNs, FCA Contexts Lattices Model)
  + Augmentation: Injection Monad.
  + Process output (ResourceURNs, FCA Contexts Lattices Model))
  + Output
  + Resolve CDI pool (signatures / annotations / subscriptions) dependencies and inject. Scopes, context, types, roles, instances, properties, functional / values predicates hash (SemanticHashing) matchings.
  + Service APIs: MapReduce. Injection. OpenRDF Elmo / AliBaba.

References:

DDD: Domain Driven Development.

DCI: Data, Context, Interaction

CDI: Context and Dependency Injection

* Containers. Deployments.
* ServiceMix deployment. Application Runtime. Events Sourcing.
* Github OSGI RDF4J Karaf Bundle. Layers Sails Services Interfaces.
* Project Parent POM. Core / Layers Bundle Modules POMs.
* Core Bundle: Layers DataFlow Model Routes. CDI. DSLs (Transforms).
* Layers Bundles (Sails Services):
* DDD, DCI, MVC, DOM, CDI. Activators. Layers Declarative Functional Uniform APIs.
* Models
* Input: Matching
* Process input: Transforms
* Augmentation: Monad.
* Process output: Transforms
* Output: Matching
* Augmentations Stack: Nested / Recursive: (DDD (DCI (MVC (DOM (CDI (DOM… (RDF4J Sail layers).
* Dispatch: CQRS via type / object / command / predicate / context (CDI) case matching. Protocol: bidi dialog prompts. ActiveMQ. Camel. Naming: DIDs, metadata.
* Augmentations Layers:
* Layers Models Builders: Templates / Transforms. Event driven (SAX like) parsers to / from layers models. Uniform Representations across layers, Layers specific Object Models.
* DynaClass/DynaBean interface from commons-beanutils
* Src (bundles). Docs. Lib (Jars, MVN, archetypes). Nashorn (Vert.x). MetaModel / Teiid. Any23. Stanbol.
* Semantic Hashing: embed entities (data, schema: information, behavior: knowledge) contexts and functional Relationships, Inferences and Protocol, Gestures, Actions encoded into Augmented Ontology identifiers.
* Semantic Hashing: W3C DIDs (Distributed Identifiers) of Models CRUD and Purpose behavior instances: DLT, Events Sourcing.
* Semantic Hashing: Declarative Services (OSGI) / Contexts Roles Interactions (Model hash resolver) Case Matching. OGM DOM / DCI Augmented Models Services. Facades.
* Semantic Hashing: Declarative Purpose (Context) / Gestures (Actions) / Steps (prompts / dialog) : Flow Protocol Abstractions. OGM DOM MVC DCI CDI DDD.
* Components, Layers: Stacks Contexts CDI: hashing resolution. Applications Domains: integration / interoperation / matching.
* Semantic Hashing: Incorporate Domains: What a given Application Domain is about to another Application Domain (a is Customer in ERP, is Employee in Payroll). Discovery: Models, Services. Employee with good performance has discounts as a Customer.
* Semantic Hashing: Layers / Stacks encoding. Models /Domains / Protocols. MVC / DCI HATEOAS Uniform ResourceURN encoded Quads REST Facades.
* Semantic Hashing: Layers, Models. Encodings: Representations, Protocols, Facades.
* Type Inference
* HKT: Monads (Kinds), Transforms (Functions) from instance data.
* Augmentations pipelines: Models Monads Layers plus Service APIs.
* Augmentation: Input / Output Quad Monad. DDD.
* Service APIs: Resources Events Dataflow.
* Model: Plain RDF. CSPOs (Layers Representation Encoding).
* Augmentation: Aggregation Monad. DCI.
* Service APIs: Aggregation. Registry. Contexts. Available Contexts Interactions.
* Model: Sets. Resources, Kinds, Contexts Model.
* Augmentation: Alignment Monad. MVC.
* Service APIs: Alignment. Naming. Interactions (roles). Available Interactions Data (players)
* Model: ResourceURN. Objects / Graph Model.
* Augmentation: Activation Monad. DOM.
* Service APIs: Activation. Index. Data (players). Available Data (player) Contexts.
* Model: OGM / DOM Objects. Objects / Graph Model.
* Augmentation: Injection. Map Monad. CDI.
* Service APIs: MapReduce.
* Model: FCA Contexts Lattices. ResourceURN URN bitstring Encoding.
* Modules. Packages, Layers DataFlow:
* CDI
* Hash Signatures. Declarative Model driven Subscriptions.
* Augmentation Sail Layers DataFlow.
* Subscriptions Bindings.
* From DDD Endpoints / Facade Requests / Events DataFlow until OGM, Semantic Hashing CDI: populate bindings (CDI), augment models and populate response parsing model driven DDD hierarchy stack and response event.
* CDI: Injection. OpenRDF Elmo. Beans Serialization. Templates: case matching. Map: FCA Contexts Lattices Model. Augmentations: Hashing (signatures / subscriptions) injection.
* FCA Model: Contexts. Map Monad.
* DOM: Models. Data.
* DOM Model: Index. Activation Monad.
* Object  (Object, Attribute(Type, Object))
* Type : Object
* Attribute : Object
* MVC: View. Interactions.
* Model: ResourceURNs. Naming. Alignment Monad.
* Model : Naming
* View : Index
* Controller : Registry
* DCI / ESB: Use Cases. Controller.
* Model: Sets. Registry. Aggregation Monad.
* Actor : Roles
* Purpose: Context
* Gestures: Roles resolution
* Actions: Interaction Steps (further matching Actions Purposes)
* DDD
* Model: RDF. Input / Output Monad
* Models Sail Layers Models Services APIs HATEOAS Protocols (REST APIs).
* Representations / Transforms Encoding: Augmentation Sail Layers Models / Models Services.
* REST Representations: Augmentations Layers Encoded Models. Protocol. Purpose, Content Type (Model Sail Layer case matching) reactive dialogs.
* Input / Output
* DataFlow: Input / Output:
* DDD REST Endpoint. Message (Events) Interaction I/O Representations.
* Augmentations:
* DDD. I/O: Input / Output. CSPO Model. Dispatch current Event (case matching) to corresponding Augmentation layers. Representations / Transforms.
* DCI / ESB. Aggregation. Registry / Context (roles). Model: Sets. Available Contexts Interactions: CDI Contexts.
* MVC. Alignment. Naming / Interactions (roles). Model: URNs. Available Interactions Data (players).
* DOM. Activation. Index / Data (players). Model: DOM. Available Data (State / Players) Contexts.
* CDI: Injection. OpenRDF Elmo. Beans Serialization. Templates: case matching. Map: FCA Contexts Lattices Model. Augmentations: Hashing (signatures / subscriptions) injection.
* Predicate Calculus Inferences: States / Order.
* Contexts:
* Resources : Data Statements (CSPO);
* Kinds : Schema Patterns (C,SK,PK,OK);
* Contexts : Transforms Patterns (C, (P, SK), (SK,OK), (P, OK));
* Selectors (context roles). Bind, unit. Map, flatMap. Transforms (domain / range) interaction flows. Functional DCI: Wrappers hierarchy (i.e.: root DOM / Resource / Kind / Contexts).
* DCI: actions dynamic DSL. Parse domain / roles (contexts selectors). DataFlows from models Contexts Transforms domains / range.
* DCI Roles: Monads. Type classes. Traits. Implicits.
* DCI Contexts / Interactions: Roles Resolution (ctx init / dialog / prompts), Functional DataFlow.
* @FunctionalInterface Parameterization. Higher Kinds Types. Case Matching: Kind (class) Role (metaclass) functions contexts arguments resolution.
* A trait, which is the type class.
* Type class instances, which are implicit values.
* Type class usage, which uses implicit parameters.
* Role streams, observable. RxJava. Vert.x.
* Higher Kinded Types. Free Monads / DSLs from quads / triples (wrapper, wrapped, ids: encoding / matching, transformations)
* Models: OGM DOM Domain Facade
* Functor Contexts / Transforms (Models, Ontology)
* Services / Augmentations DataFlows.
* Data, Context, Interaction DataFlows.
* HATEOAS / HAL Model, View, Controller
* DDD CDI Connectors (Protocols)
* Data, Information, Knowledge:
* OGM DOM. MVC / DCI. CDI DDD.
* Model / Data: OGM / DOM (Resource, Instance).
* Use Cases / Contexts / Controller. MVC / DCI (Class, Metaclass).
* Roles / Views / Interactions: CDI / DDD (Occurrence, Role).
* Use Cases / Data Model / Roles Interactions:  ResourceURN aggregated role Statements pairs.
* Data: Resource (actor), Instance (occurrence).
* Contexts: Class (player), Metaclass (role).
* Interactions: Occurrence, Role.
* In DCI, a Context holds the mappings from roles in an interaction to specific object instances, and also have the interactions that can be performed on this mapping.
* Finally, interactions are implemented as simple methods in the context. They may take arguments, and then look up mapped roles in the context map, and fire off a domain method. The important point here is that the interaction methods should match whatever actions you have in your user interface, so that the code matches the users mental model.
* We have three entities, Project, User, and Task. Tasks implement the Assignable role. Project and User implement the Assignments role. Users can be Assignees. We then have a main context with interactions called InboxContext.
* Both Users and Projects have an Inbox with tasks, and it is these that we want to assign to Users. But they can be "owned" either by the User itself, or by a particular Project, hence the need for a separate role Assignments so that our notion of "assignment" is not tied to "given a users list of tasks, one of them can be assigned to the user" but rather "given a Assignments collection of Assignables, pick one and assign to an Assignee". This way the interaction and context is entirely separated from the actual classes, and the only thing we need are the appropriate roles.
* Executing an interaction
* Let's see what happens as we walk through an execution of the "assign" interaction. The first thing we need to do is set up the context:
* InteractionContext map;
* RootContext context;
* InboxContext inboxContext;
* I create a new context map for this interaction, and instantiate a new RootContext. This symbolizes the root of all contexts in my application, and will mainly hold methods for getting to subcontexts. I pass in the map so that the RootContext can pass it on during the user() call, which will create the subcontext UserContext that has all the interactions and subcontexts for working with a selected user.
* I pass in the userId so that the user() method can do the lookup. If this DCI implementation is used in a REST API setting that context lookup will basically map to the URL, so the "userId" will be one part of the URL being referenced. You can imagine the above being mapped to "/administrator/inbox" in a URL.
* The user() method will add the given user to the context map, and then create a new subcontext with the extended map. Here's what it looks like:
* public class RootContext   extends Context
* public UserContext user(String id)
* "Context" is a baseclass that has the InteractionContext in a variable "context", and a "subContext" method for easily instantiating new contexts with that context map. What I do here is to look up the UserEntity with the given id from the Qi4j UnitOfWork, and then register it in the map with the given roles. The object already has those roles, so the only thing that happens here is that the map knows that if someone asks for the object playing the "Assignee" role, it knows what to return.
* Once the context has been looked up it is time to invoke the interaction, with arguments:
* [inboxContext.assignTo](http://inboxcontext.assignto)( task );
* Since the context has access to the context map the above method doesn't need to know who to assign it to. That is given by the context map! The implementation of assignTo() is as follows:
* public class InboxContext   extends Context {
* public void assignTo( Assignable assignable )
* {
* [context.role](http://context.role)( [Assignments.class](http://assignments.class)).assignTo( assignable, [context.role](http://context.role)( [Assignee.class](http://assignee.class) ));
* }
* }
* The assignTo() interaction uses the context map to look up the objects bound to the roles Assignments and Assignee, and invokes the assignTo method given these objects. In the above you therefore see exactly how the context, interaction and roles interact to implement a given usecase.
* class AssignmentsMixin  implements Assignments
* @This AssignmentsData data;
* public void assignTo( Assignable assignable, Assignee assignee )
* {
* [assignable.assignTo](http://assignable.assignto)( assignee );
* [data.assignments](http://data.assignments)().add( assignable );
* }
* public Iterable<Assignable> assignments() {
* return [data.assignments](http://data.assignments)();
* }
* The AssignmentsMixin implementation assigns the Assignable to the Assignee, and then adds it to the list of assignments. Nowhere in this code do we see that we are talking about Users, Tasks or Projects. It is all related to the roles related to handling assignment. This allows us to focus on one thing at a time, and makes it clear what the boundaries are between various algorithms and roles that interact in our system as a whole.
* The @This injection is what provides the private mixin support. The field will be injected with a reference to "this object", cast to the  "AssignmentsData", which is a mixin that holds the data for managing  assignments. This cannot be reached from the outside of the entity, however. What we want is to ensure that all access to data of entities are accessed through our roles. This helps keep our state encapsulated.
* If another algorithm also needs to use the same state, then all it has to  do is perform the same injection. This way that other functionality, for some other usecase, can be kept separate from this AssignmentsMixin, so that each mixin deals with one thing, and one thing only.
* Now that you know how to assign a task to a user, let's try switching things around: a task will be assigned to a user, but within the context of a specific project. The code to do this looks like this:
* InteractionContext stack = new InteractionContext();
* RootContext context = [assembler.objectBuilderFactory](http://assembler.objectbuilderfactory)().newObjectBuilder( [RootContext.class](http://rootcontext.class) ).use( stack ).newInstance();
* [context.user](http://context.user)( [user.identity](http://user.identity)().get() ).project( [project.identity](http://project.identity)().get() ).inbox().assignTo( task2 );
* In this case, instead of letting the user play both the Assignee and  Assignments roles, the user will only be used for the Assignee. The  project which is looked up in the project() call will be bound to  Assignments, so that once we get to assignTo() in the InboxContext, the  algorithm will essentially say: "Assign the Task to the User within the  Projects assignment". And to do this we did not have to change any code in the assignments handling. The only thing that changed was what  objects were bound to what roles! Let me hear you say: "SWEET!"
* What we have seen here is a simple example of how DCI can be implemented in Qi4j, and which provides all the key ingredients needed: Roles, Data, Contexts and Interactions.
* Interactions
* So where does the use-case-specific code go? The answer is: in the object roles. The concept of roles is pretty unique to DCI – I haven’t seen it in any other major architectural style. The roles are supposed to dynamically extend the data object’s behavior with the use case-specific functionality. Since such functionality might want to operate on object’s state (like the printer above), an object role should preferably have access to the object’s internals e.g. via accessors or simple methods
* Class Book
* Role PrintedBook
* Book justData : [books.findByID](http://books.findbyid)(id);
* PrintedBook youCanPrintMe : justData extendedBy PrintedBook;
* Second, an object role can be played by objects of different classes, as long as they contain data and methods necessary to fulfill the role. In our example, we could print a magazine the same way we print a book (we should adjust the role‘s name then).
* Context
* The place where data objects are retrieved and roles assigned is called the context. This would be a rough equivalent of an application service or Clean Architecture’s use case interactor. Why only rough? Because ideally, a context should provide the roles with references to all collaborators in a use case, call one of the roles and do nothing else.
* Think about a bank transfer. We have two accounts – source account and destination account. Obviously, the account is a data object, while source account and destination account are roles. When the source account decreases its own balance, it wants to let the destination account know that it should increase its balance.
* DCI assumes that object roles should know what collaborators they have based on the context in which they execute.
* All Together
* Let’s walk through things the way control flows in the application. A user presses a button that sends a use-case-related request. An application receives that request, instantiates an appropriate context, and passes the request to it. The context retrieves all data objects necessary to fulfill the use case‘s goal and assigns appropriate roles to them. Then, it sends a message to an object role that begins a series of interactions between the objects. If there is a need to communicate something to the user, it’s also done by the roles. Once it’s complete, the user’s goal should be achieved.