Domain-Driven Design Tackling Complexity in the Heart of Software by Eric Evans

Chapter 1: Crunching Knowledge

Ingredients of effective modelling

1. Binding the model and the implementation.
2. Cultivating a language based on the model.
3. Developing a knowledge-rich model.
4. Distilling the model.
5. Brainstorming and experimenting.

Knowledge Crunching

- Crunching is not a solitary activity, it is need to be done as a team of developers, domain experts.
- Should have provision to provide feedback – opportunity to learn from programmers to gain experience with early version of software.
- Developer needs to gain domain knowledge – that way project ca be bring into a point where, when a powerful new feature unfolds as a *corollary* to older feature.
- Abstraction needs to achieved in collaboration with the domain-experts else the reach remains shallow.
- Need to have establish a communication between domain-experts and developer, as domain-experts distil their domain and evolves their models, the same refinement need to be understood by the developers.
- The model needs need to made simple to implement and understand.

Constant Learning

Highly productive teams grow their knowledge consciously, practicing continuous learning (Kerievsky 2003). For developers, this means improving technical knowledge, along with general domain modelling skills.

Knowledge Rich Design

A domain model is beyond entities and values that knowledge crunching gets intense, because there may be actual inconsistency among business rules. Domain experts are usually not aware of how complex their mental processes are as, in the course of their work, they navigate all these rules, reconcile contradictions, and fill in gaps with common sense. Software can't do this. It is through knowledge crunching in close collaboration with software experts that the rules are clarified, fleshed out, reconciled, or placed out of scope.

Deep Model

Useful models seldom lie on the surface. As we come to understand the domain and the needs of the application, we usually discard superficial model elements that seemed important in the beginning, or we shift their perspective. Subtle abstractions emerge that would not have occurred to us at the outset but that pierce to the heart of the matter.

New understanding keeps getting added to the base model (model that was developed at the beginning), as the model evolves this change the model profoundly, because of the knowledge crunching which is a never-ending process.

Chapter 2: Communication and the use of Language

A domain model can be the core of a common language for a software project. The model is a set of concepts built up in the heads of people on the project, with terms and relationships that reflect domain insight. These terms and interrelationships provide the semantics of a language that is tailored to the domain while being precise enough for technical development. This is a crucial cord that weaves the model into development activity and binds it with the code.

Developer develops their own technical jargons and domain-experts have their own jargon which they usages in their field, the linguistic divide can lead to multiple inconsistency – even if there are few people who are bilingual and able to communicate in both developers and the domain-expert languages, it is not enough.

Hence a base model is required, the base model should allow developers and domain experts to communicate freely, the base model should supply the language for developers and domain experts to communicate with each other, and for the domain experts to communicate among themselves about requirements, development planning and features. The more pervasively, the language is used, the more smoothly understanding will flow.

Initial model, would have limitation as there would be some degree of adulteration and may contains ambiguous and contradictory terms. Constant use of the model / language will expose these gaps, which can then be overcome.

Modelling out Loud

To build and effective domain model, it needs to be used in every day communication which will highlight the gaps, which needs to addressed and continued to use the domain model for communication. IN agile process as the requirements evolved, the domain language should need to be refined to meet the new requirements.

With a UBIQUITOUS LANGUAGE , conversations among developers, discussions among domain experts, and expressions in the code itself are all based on the same language, derived from a shared domain model.

Documents and Diagrams

Though UML diagram are widely used, they failed to deliver to important aspect of a model : the meaning of the concept it represents and what the object are meant to do.

A detailed diagram would be to overwhelm to read, hence its better to write a document illustrated with specific and simplified diagram.

Document need to remain current, even sometime the written code does not convey the concept the written line. Hence a salified specification document needs to be maintained which needs to supplement by detailed scenario-based documents.

A domain model comes to reflect the most relevant knowledge of the business, application requirements documents become scenario within the domain model & can be used to describe the scenarios using model-driven-design.

One model should underlie implementation, design, and team communication. Having separate models for these separate purposes poses a hazard.

Chapter 3: Binding model and implementation

Domain Model

- Domain model should consider the solution the solves the problem, at the same time honour the software design principles.
- Domain model should clearly express the key design concept of the domain.
- Domain model should be practical to implement.
- Domain model should reflect the design of the system, and its implicit mapping.
- Domain model should support UBIQUITOUS LANGUAGE.
- Only one model should address a particular part of the system.

Domain driven design does not differential between analysis model and design model, it believes in a single model can server both these purposes.

For domain driven design, object-oriented language is more preferred – procedural language have limitation as they cannot have a high level grouping beyond functions, whereas in object oriented languages it comes by default as they are centred around objects which can be used to group functionality and have a well define relationship beyond procedural language functions.

The following are the advantage of using object paradigm for domain driven design.

- Object oriented language use of objects naturally have **Intrinsic advantage.**
- As model needs to be worked with other models – Object oriented language have proven a well-established way to have objects easily integrated with other objects of other models. Making its easy to work with subsystems following other paradigms.

Chapter 4: Isolating the domain

Its difficult to represent a complete system in a single model, hence its required to separate out the model into sub models , which needs to be grouped as per high level functionality.

well-isolated domain layer allows developers to focus on the intricacies of the domain model without being distracted by the complexities of software technology or other system functions.

- Enables a clean separation of concerns, making each layer of the system easier to understand and maintain.
- Reduces maintenance costs, as layers tend to evolve at different rates and in response to different needs.
- Facilitates deployment in distributed systems by allowing layers to be flexibly placed on different servers or clients

A layer architecture – splits the system into separate layer, for improving understanding and maintainability of the system.

Usually, a system can be spitted into following layers

- **UI Layer** – presentation layer
- **Application Layer** – *Thin layer that helps business to interact with application layer of the other system. It does not contain any business rules, knowledge or state reflecting business situation but it can contain state that reflects the progress of a task for the user or the program.*
- **Domain Layer –** *This layer is responsible for the business, information about the business situation, and business rules.* ***This layer also where the domain model lives****.*
- **Infrastructure Layer** – This layer provides the necessary support to above layers.

Layered architecture helps in segregation of concerns – allowing domain objects from free itself from the responsibilities of displaying themselves, string themselves, managing application tasks and other responsibilities, so that it can focus on implementing the business functionalities.

Separation of layer allowing each layer to manage its responsibilities better, with separation of concerns. Even if the layers are separated, they need to be worked together, without losing the benefit of the separation is the motivation behind number of patterns that were developed over the years.

- Model view controller pattern, initially adapted for smalltalk in 1970's
- Model view separation pattern, 1998 is also based on the similar concept.

These patterns were developed allowing domain model to be developed, without simultaneously thinking about user interface that interacted with it.

Architectural framework

Architectural framework plays an important role in software designing – it should not be making too many assumptions that constrain the design choices, or should not be making the implementation so much heavy weight that it slows down the development.

When choosing an application framework – team need to explore two things

1. Building an implementation that express a domain model.
2. Application model need to be used to solve an important problem.

Application team need not to leverage all the feature available within the framework; if the above two concerns are addressed.

Domain Layer

Smart-UI is an anti-pattern, it does not align itself with the domain model paradigm. There are specific scenarios, where the requirements are modest and timeline are very tight, in such projects it makes sense to adapt this anti-pattern.

Smart-UI is best fit for

- Simple application.
- Developer maturity is low, application required less capable developer to deliver work with little training.
- Even deficiencies in requirements analysis can be overcome by releasing a prototype to users and then quickly changing the product to fit their requests.
- Applications are decoupled from each other, so that delivery schedules of small modules can be planned relatively accurately. Expanding the system with additional, simple behavior can be easy.
- Relational databases work well and provide integration at the data level.
- 4GL (*fourth-generation language, tools are programming languages that are designed to be easier for users to develop applications*) tools work well.
- When applications are handed off, maintenance programmers will be able to quickly redo portions they can't figure out, because the effects of the changes should be localized to each particular UI.

There are other kind of architecture between perfectly aligned layered architecture and smart-UI, like transaction script – this architectural pattern segregates UI from application but does not isolate the domain related code from the rest of the application system logic.

Model-driven-design is very much needed for a complex application.

Other kind of isolations

There are other domain components that may not follow the domain model, so one need to have the boundaries define to ensure the domain model is not corrupted – having a Bounded Context define and placing an Anti-corruption layer in between, one can ensure clear distinction within the domain model.

Chapter 5: A Model Expressed in Software

Domain-Driven Design focuses on the essential patterns used to represent a domain model within software. These building blocks help translate conceptual understanding into concrete code elements that remain connected to the domain. The chapter emphasizes that implementation choices should be driven by insight into the domain, ensuring that the software accurately and effectively reflects the business logic.

Defining association between object of a domain model is a difficult task – ensuring the same is not lost during implementation is another challenge within itself.

There are three different model elements

- **Entities (Reference Objects)** – objects that represents something with continuity (constantly without breaking its integrity over time) and identity are called as entity. These objects have a distinct identity that persists regardless of changes to their attributes.
- **Value Objects** – the attributes that describes the state of something is called as value object.
- **Services** - Services are the action or operations that are performed. *A SERVICE is something that is done for a client on request.*
- **Module –** This are parts of the domain, which should reflect a concept within a domain. It should not be treated as a mere technical container , they reflects meaningful domain concepts.

Association - For every traversable association in the model, there is a mechanism in the software with the same properties.

The following are the three ways to make association more tractable (easy to control or influence).

1. ***Impose a traversal direction***
2. Add a qualifier, effectively reduce multiplicity.
3. Eliminating nonessential association.

A bidirectional relation means, the objects can be understood only together. If the application does not call for traversal in both directions, then adding a bidirectional traversal relation reduce impendence and simplicity of the design. Understanding the domain may reveal a natural directional bias. Ultimate simplification is to remove the association within the objects. Simplifying associations, even at the cost of adding extra objects, is vital for clarity and practicality

An ENTITY is anything that has continuity through a life cycle and distinctions independent of attributes that are important to the application's user. On the other hand, not all objects in the model are ENTITIES, with meaningful identities. An Entity can have many attributes, one need to include those attributes that are required the behaviour.

Each ENTITY must have an operational way of establishing its identity with another object–distinguishable even from another object with the same descriptive attributes.

ENTITY should be uniquely identifiable, either by an attribute or set of attributes. Defining identity demands understanding of the domain. TO identify an entity uniquely, sometimes unique symbols are attached to it which can clearly identify an entity within a system (the symbol acts like an ID for the entity). Often the ID is generated automatically by the system. The generation algorithm must guarantee uniqueness within the system, which can be a challenge with concurrent processing and in distributed systems. An ID should be able to uniquely identify an entity within a system, in some case even outside the system.

An object that represents a descriptive aspect of the domain with no conceptual identity is called a VALUE OBJECT. VALUE OBJECTS are instantiated to represent elements of the design that we care about only for what they are, not who or which they are. A value object and entity are interchanging, depending on the scenarios. A VALUE OBJECT can be made up of other VALUE OBJECT(s). VALUE OBJECT can even have reference to ENTITIES.

Value Objects are elements within a software model that describe the state of something else. Unlike **Entities**, which have a unique identity, Value Objects are defined solely by their attributes. Think of them as attributes or characteristics that provide context or further describe other elements in your model.

Examples of value object

- **Money**: Instead of representing money with number, define it as a value object which will have attribute like Amount and Currency.
- **Address**: An address is a value object that have Street, Zip, City, County as attributes.
- **Colour**: Colour is a value object which holds RGB values.

Key characteristics of Value Objects:

- **No persistent identity**: A value object is defined by its attribute; they do not have any unique identity. A money value object with same amount and currency value ,is consider equal and interchangeable regardless of what system it's been referred.
- **Immutability**: Once created, a value object's attribute should not change. If one need to change its attribute it would result in creating a new value object instead.
- **Replaceability**: Since the value object don't have their own identity , they can be easily replicable y another value object which has same attribute values.

Advantage of using Value Objects

- **Improve code readability and maintainability:** Value Objects encapsulate related data, making your code easier to understand and work with.
- **Reduce errors**: Enforcing immutability and well-defined behaviour within Value Objects helps prevent inconsistencies and bugs in your application.

- **Enhance domain expressiveness**: Value Objects allow you to model domain concepts more accurately and expressively, leading to a better reflection of the business logic in your code.

Copying vs Sharing objects

In case of copying, a same object is copied from one system to another system, single copy lives in both the systems independently. This may clog the systems

In case of sharing a same object is shared across two systems, where a single copy is shared between the systems. It required messages to be send back to object for each transaction so that object can be modified accordingly.

Sharing is more preferred under following conditions

- When saving space or the object count in the database is critical.
- When the communication overhead is low.
- When the shared object is strictly immutable.

Service

Operation that does not belong to Entity or a value object, should be categorized as SERVICE. A SERVICE is an operation offered as an interface that stands alone in the model, without encapsulating state, as ENTITIES and VALUE OBJECTS do. SERVICES are a common pattern in technical frameworks, but they can also apply in the domain layer.

Service operation names should come from the UBIQUITOUS LANGUAGE or be introduced into it. Parameters and results should be domain objects.

A good Service has following characteristics

1. The operation relates to a domain concept that is not a natural part of an ENTITY or VALUE OBJECT.
2. The interface is defined in term of other elements of the domain model.
3. The operation is stateless.

Example of a Service: In a banking application, transactions can be exported into a spreadsheet for analysis, the EXPORT is a service which otherwise do not have any significance in the bank's domain model.

A service can be any layer within a domain

a) Application Layer Service Example: Fund transfer App Service.
    - Digest input (inputs in XML or any other canonical format)
    - Send message to the domain service for fulfilment.
    - Listen for confirmation.
    - Decides to send notification using infrastructure service.
b) Domain Layer service: Fund transfer Domain Service
    - Interacts with the necessary Account and Leger objects, making appropriate debit and credits.
    - Supplies confirmation of results (transfer allowed or not allowed)
c) Infrastructure Layer service: Send Notification
    - Send email, letters and other communication as directed by the application.

Elaborate architectures should be used only when there is a real need to distribute the system or otherwise draw on the framework's capabilities for interfacing with domain services.

Modules (a.k.a. Packages)

Modules should be a meaningful part of the domain model, reflecting concepts within the domain.

The flowing are the key points related to a module

- **Low Coupling and High Cohesion**: There should be low coupling between modules and high cohesion within them. Low coupling minimizes the overhead of understanding relationships between modules, while high cohesion brings together elements with rich conceptual relationships.

- **Conceptual Grouping**: Modules should group concepts, not just code.
- **Modules as Chapters** the model is telling a story, modules are like chapters. Module names should be part of the ubiquitous language.
- **Co-evolution**: Modules and smaller elements should co-evolve. However, modules tend to be conceived early in a project and not changed, even when the model objects change. Refactoring modules is more disruptive but can allow more freedom for objects within them to evolve.
- **Communication Mechanism**: Modules are a communication mechanism.
- **Avoid Technical Partitioning:** avoid packaging schemes driven by technical concerns, such as separating objects by persistence or layering.
- **Domain-Driven Packaging**: Domain developers should have as much freedom as possible to package domain objects in ways that support their model and design choices. An exception to this is generated code, which should be put in a separate package to avoid cluttering up the design elements that developers work with.

Modelling Package

- **Model design & Implementation Technology should be aligned**: It is important to choose the implementation technology that aligns with the modelling paradigms. Technology should not be chosen arbitrarily.
- **Object-Oriented Design:** The object-oriented paradigm currently dominates the majority of ambitious software projects. This is due to multiple factors including the suitability of objects for modelling many real world scenarios, and the wide availability of tools and resources for object oriented development.
- **Procedural Languages**: Model-driven design has limited applicability using procedural languages because they do not have a modelling paradigm that corresponds to a purely procedural language. **Procedural languages focus on a series of steps to follow rather than conceptual connections**. Although procedural languages may support complex data types, they only capture organised data, not the active aspects of the domain. Domains that are intensely mathematical or that are dominated by global logical reasoning do not fit well into the object-oriented paradigm, for them Procedural language can be an option.
- **Mixing Paradigms**: Often needs occurs when one needs to mix paradigms like Rule engine or a workflow engine with object-oriented models. **When mixing paradigms, it is crucial to maintain a single model that works with both implementation paradigms.** A robust UBIQUITOUS LANGUAGE is the most effective tool for holding the parts together in a heterogenous mode. Relational databases are a special case of paradigm mixing, being the most common non-object technology that is also intimately related to the object model because it acts as the persistent store of the data that makes up the objects themselves.

Chapter 6: The Life Cycle of a Domain Object

An AGGREGATE is a cluster of associated objects that we treat as a unit for the purpose of data changes. Each AGGREGATE has a root and a boundary.

ENTITIES other than the root have local identity, but that identity needs to be distinguishable only within the AGGREGATE, because no outside object can ever see it out of the context of the root ENTITY.

An **AGGREGATE** is a cluster of associated objects that are treated as a unit for the purpose of data changes.

- Each **AGGREGATE** has a **root** and a **boundary.**
- The **boundary** defines what is inside the **AGGREGATE**
- The **root** is a single, specific **ENTITY** contained in the **AGGREGATE**
- The **root** is the only member of the **AGGREGATE** that outside objects are allowed to hold references to, although objects within the boundary may hold references to each other.
- **ENTITIES** other than the root have local identity, but that identity needs to be distinguishable only within the **AGGREGATE**, because no outside object can ever see it out of the context of the root **ENTITY.**
- **Invariants** which are consistency rules that must be maintained whenever data changes, will involve relationships between members of the **AGGREGATE.**
- Any rule that spans **AGGREGATES** will not be expected to be up-to-date at all times.
- The invariants applied within an **AGGREGATE** will be enforced with the completion of each transaction.

To translate that conceptual **AGGREGATE** into the implementation, a set of rules must be applied to all transactions.

- The **root ENTITY** has global identity and is ultimately responsible for checking invariants
- **Root ENTITIES** have global identity. **ENTITIES** inside the boundary have local identity, unique only within the **AGGREGATE.**
- Nothing outside the **AGGREGATE** boundary can hold a reference to anything inside, except to the **root ENTITY**. The **root ENTITY** can hand references to the internal **ENTITIES** to other objects, but those objects can use them only transiently, and they may not hold on to the reference. The root may hand a copy of a **VALUE OBJECT** to another object, and it doesn't matter what happens to it, because it's just a **VALUE** and no longer will have any association with the **AGGREGATE.**
- Only **AGGREGATE** roots can be obtained directly with database querie. All other objects must be found by traversal of associations.
- Objects within the **AGGREGATE** can hold references to other **AGGREGATE** roots.
- A delete operation must remove everything within the **AGGREGATE** boundary at once.
- When a change to any object within the **AGGREGATE** boundary is committed, all invariants of the whole **AGGREGATE** must be satisfied.

Example:

- The car is an **ENTITY** with global identity.
- The tires must be identified **ENTITIES** also.
- It is very unlikely that the identity of those tires is important outside of the context of that car.
- Therefore, the car is the **root ENTITY** of the **AGGREGATE** whose boundary encloses the tires.
- On the other hand, engine blocks have serial numbers and are sometimes tracked independently of the car, and in some applications, the engine might be the root of its own **AGGREGATE.**

FACTORIES are used to encapsulate the creation of objects or entire **AGGREGATES** when the creation process becomes complicated or reveals too much of the internal structure.

- **FACTORIES** shift the responsibility for creating instances of complex objects and **AGGREGATES** to a separate object, which may have no responsibility in the domain model but is still part of the domain design.
- A **FACTORY** provides an interface that encapsulates complex assembly and does not require the client to reference the concrete classes of the objects being instantiated.
- **FACTORIES** create entire **AGGREGATES** as a piece, enforcing their invariants

Reasons to Use FACTORIES:

- **Encapsulation**: A **FACTORY** encapsulates the knowledge needed to create a complex object or **AGGREGATE**. Just as the interface of an object should encapsulate its implementation, thus allowing a client to use the object's behaviour without knowing how it works, a **FACTORY** encapsulates the knowledge needed to create a complex object or **AGGREGATE.**
- **Abstraction**: A **FACTORY** provides an interface that reflects the goals of the client and an abstract view of the created object.
- **Complexity Management**: Assembling a complex compound object is a job best separated from the object's function. **FACTORIES** are used when the creation of an object, or an entire **AGGREGATE**, becomes complicated or reveals too much of the internal structure. Shifting responsibility to the client object leads to problems because the client must know something about the internal structure of the object.
- **Enforcing Invariants**: Each creation method in a **FACTORY** should be atomic and enforce all invariants of the created object or **AGGREGATE**, ensuring that a **FACTORY** only produces an object in a consistent state. For an **ENTITY**, this means creating the entire **AGGREGATE**, with all invariants satisfied, but with optional elements still to be added. For an immutable **VALUE OBJECT**, this means that all attributes are initialized to their correct final state.

Types of FACTORIES:

- **FACTORY METHOD**: A **FACTORY METHOD** is a method on an existing object that creates another object. It is used when you want to hide the details of creating an object and the control needs to be with the object that has the

method. For example, if you needed to add elements inside a preexisting **AGGREGATE**, you might create a **FACTORY METHOD** on the root of the **AGGREGATE**. This hides the implementation of the interior of the **AGGREGATE** from any external client, while giving the root responsibility for ensuring the integrity of the **AGGREGATE** as elements are added.

- **Standalone FACTORY:** A standalone **FACTORY** is a dedicated object or **SERVICE** that is responsible for creating objects. It is used when there is no natural host for the creation process or when you want to hide the concrete implementation or complexity of construction. A standalone **FACTORY** usually produces an entire **AGGREGATE**, handing out a reference to the root, and ensuring that the product **AGGREGATE'S** invariants are enforced. If an object interior to an **AGGREGATE** needs a **FACTORY**, and the **AGGREGATE** root is not a reasonable home for it, then a standalone **FACTORY** can be created.

Choosing FACTORIES and Their Sites:

- **Control**: You place the **FACTORY** where you want the control to be. These decisions usually revolve around **AGGREGATES.**
- **Relationship**: A **FACTORY** should be attached only to an object that has a close natural relationship with the product.
- **Hiding Complexity**: When there is something you want to hide—either the concrete implementation or the sheer complexity of construction—yet there doesn't seem to be a natural host, you must create a dedicated **FACTORY** object or **SERVICE.**

When to use a Constructor:

- If you have simple objects without polymorphism, using a direct constructor may be a better choice than introducing a **FACTORY**. **FACTORIES** can obscure simple objects that don't use polymorphism.

Designing the Interface of a FACTORY

- **Atomicity**: Each operation of a **FACTORY** must be atomic. You must pass in everything needed to create a complete product in a single interaction with the **FACTORY**. You also must decide what will happen if creation fails, in the event that some invariant isn't satisfied.
- **Coupling**: The **FACTORY** will be coupled to its arguments. If you are not careful in your selection of input parameters, you can create a dependencies. The safest parameters are those from a lower design layer.

ENTITY FACTORIES vs. VALUE OBJECT FACTORIES:

- **VALUE OBJECT FACTORIES** produce immutable objects, with all attributes initialised in their final state and can be fully described in the **FACTORY.**
- **ENTITY FACTORIES** tend to take only the essential attributes required to make a valid **AGGREGATE**. Details can be added later if they are not required by an invariant. They also have to deal with assigning an identity to the ENTITY, which is irrelevant to **VALUE OBJECTS.**

Example of FACTORY:

- A **FACTORY** can be used to create a Cargo object. There could be a newCargo method on the **FACTORY** that takes Cargo prototype and a newTrackingID, it would return a Cargo with an empty Delivery History, and a null Delivery Specification.

Reconstitution of Objects:

- Whenever there is exposed complexity in reconstituting an object from another medium, the **FACTORY** is a good option.
- **FACTORIES** can be used to reconstitute objects from databases or XML, and object-mapping technologies may provide some of these services.

Relationship with REPOSITORIES:

- A **FACTORY** handles the beginning of an object's life, while a **REPOSITORY** helps manage the middle and end.

- **REPOSITORIES** may delegate object creation to **FACTORIES** when reconstituting stored objects.

Key Takeaways:

- **FACTORIES** are essential for managing the complexity of object creation in a domain-driven design.
- They enforce invariants, encapsulate creation logic, and provide an abstract interface for clients.
- **FACTORIES** are a part of the domain design that helps keep the model-expressing objects sharp.

Repositories

A **repository** is a mechanism for encapsulating storage, retrieval, and search behaviour, emulating a collection of objects. Think of a **repository** as a specialised intermediary between your software's domain objects and the data storage mechanism, like a database. It acts as an in-memory collection of all objects of a certain type, but with added querying capabilities. It provides an interface to access persistent objects, managing their life cycle, while hiding the complexities of data storage. Instead of directly accessing a database, you use a repository to manage your objects, similar to a library managing books, but for software objects.

Why Use a Repository?

- **Simplified Object Access**: Repositories offer a simplified model for obtaining persistent objects, abstracting away the details of the underlying data store. Clients can request objects using a clear interface based on the model.
- **Decoupling**: They decouple the application and domain design from specific persistence technologies, multiple database strategies or data sources. This makes it easier to switch data stores without changing the application's core logic.
- **Design Communication**: Repositories communicate design decisions about object access clearly. They make explicit which objects should be accessed directly and which should be accessed through traversal.
- **Testability**: They allow for easy substitution of a dummy implementation (often an in-memory collection) which is beneficial for testing.
- **Focus on the Model**: They help keep the client focused on the domain model, by delegating all object storage and access to the repositories.

Key Features of a Repository:

- **Abstraction**: A repository hides the inner workings of storage, retrieval and querying, making the client code simpler.
- **Collection-like Interface**: It acts like a collection, providing methods to add, remove, and retrieve objects.
- **Querying**: It provides methods to search for objects based on criteria. This could be based on attribute values or other selection criteria.
- **Encapsulation**: It encapsulates the machinery of database queries, and any metadata mapping.
- **Global Access**: It provides a well-known global interface to access the roots of **AGGREGATES** that need direct access.

When to Use a Repository:

- Repositories are primarily for **AGGREGATE** roots that require direct access. These are usually **ENTITIES** but sometimes include **VALUE OBJECTS** with complex internal structures, or enumerated **VALUES**.
- They are not intended for objects that are internal to an **AGGREGATE**, as these should be accessed through traversal from the root.
- They are not needed for transient objects that are used and discarded.

How a Repository Works:

- **Client Request**: A client requests an object from the **repository**.
- **Query Execution**: The **repository** uses a query method to locate the object or objects.
- **Retrieval**: The **repository** retrieves the object from the data store using the necessary infrastructure services.
- **Object Instantiation**: The **repository** instantiates a fully formed object and returns it to the client.
- **Persistence**: If the object is new, the **repository** will handle the insertion of that object to the data store.

Types of Queries:

- **Hard-Coded Queries**: These are specific queries with particular parameters, built directly into the repository. They can be used to retrieve an entity by its identity, or to retrieve a collection based on attribute values, value ranges, or complex combinations of parameters. They can also perform summary calculations7.
- **Flexible Queries**: These use a framework to allow clients to specify query criteria without needing to know how they will be implemented. One way to achieve this is through **SPECIFICATION**-based queries. A **SPECIFICATION** allows a client to specify *what* they want, without specifying *how* to get it.

Implementing a Repository:

- **Abstraction**: The implementation should hide the underlying persistence technology from the client.
- **Adaptability**: The concept can be implemented with various storage mechanisms (object database, relational database, in-memory storage).
- **Delegation**: The **repository** delegates to the appropriate infrastructure services for storage and retrieval.
- **Transaction Control**: The **repository** does not typically handle transaction control, leaving that responsibility to the client.
- **Type Abstraction:** A single **repository** can manage multiple classes, using abstract superclasses, interfaces, or concrete classes. You may be constrained by the lack of polymorphism in your database.

Key Considerations:

- **Performance**: Developers need to be aware of the performance implications of different repository implementations and query methods.
- **Avoid 'Find or Create' Functionality**: The repository should not mix the creation and retrieval of objects to avoid confusion. If a client needs a **VALUE OBJECT** they should go directly to a **FACTORY**.

Relationship with other concepts:

- **FACTORIES**: A **FACTORY** handles the beginning of an object's life cycle, while a **repository** manages the middle and the end. **FACTORIES** create new objects and reconstitute stored objects. **REPOSITORIES** find existing objects.
- **AGGREGATES**: **REPOSITORIES** provide access to the roots of **AGGREGATES**, which encapsulate a group of associated objects.
- **SPECIFICATIONS**: **SPECIFICATIONS** can be used with **REPOSITORIES** to define flexible query criteria. They allow the client to describe *what* they want to retrieve, without specifying *how* to retrieve it.

Designing Objects for Relational Databases

The most common non-object component in primarily object-oriented software systems is the relational database. This presents challenges because it involves a **mismatch of paradigms**. Relational databases store data in tables with rows and columns, while object-oriented systems work with objects that have attributes and behaviours. This mismatch can impact the object model. The database is not just interacting with the objects; it is storing the persistent form of the data that makes up the objects themselves.

Three Common Scenarios

There are three common scenarios when dealing with relational databases in the context of object-oriented systems:

- The database is primarily a repository for the objects.
- The database was designed for another system.
- The database is designed for this system but serves in roles other than object store.

Database as a Primary Repository

When the database schema is created specifically to store objects, it is beneficial to accept some model limitations to keep the mapping simple. This approach prioritises a tight coupling between the model and implementation. Here are some points to consider:

- **Simple Mapping**: Without other demands on schema design, the database can be structured to make aggregate integrity safer and more efficient as updates are made.
- **Transparency**: The mappings between objects and tables should be transparent and easy to understand by inspecting the code or reading the mapping tool entries.
- **Trade-offs**: Some richness of object relationships may be sacrificed to keep closer to the relational model.
- **Compromise**: It may be necessary to compromise some formal relational standards, such as normalisation, if it helps simplify the object mapping.
- **No External Access**: Processes outside the object system should not access such an object store to avoid violating the invariants enforced by the objects. Their access would also lock in the data model and make refactoring the objects more difficult.
- **Direct Mapping**: A table row should contain an object, perhaps along with subsidiaries in an **AGGREGATE**. A foreign key in the table should translate to a reference to another **ENTITY** object.
- **Avoid Divergence**: Don't allow the data model and object model to diverge significantly, regardless of mapping tool capabilities.

Database Designed for Another System

In many cases, the data comes from a legacy or external system not intended as an object store. This means two domain models may coexist within the same system. In such situations, it might be better to conform to the model implied by the other system, or to keep the models completely distinct.

3. Database Designed for the System but with Multiple Roles

Sometimes, the database might be designed for the system, but also serve other roles beyond object storage. This can lead to a schema that is quite different from the object model.

- **Separate Schemas**: Cutting the database loose from the object model is a tempting path. If chosen consciously it can result in a clean database schema not constrained by the object model.
- **Other Software**: The database may also be used by other software that does not use objects.
- **Rapid Evolution**: The database may require little change, even when the behaviour of objects changes rapidly.
- **The Risk**: A team may fail to keep the database current with the model.

The Importance of a Ubiquitous Language

The **UBIQUITOUS LANGUAGE** should tie the object and relational components together as a single model. The names and associations of elements in the objects should meticulously correspond to those of the relational tables to avoid confusion. Although mapping tools might make this seem unnecessary, subtle differences in relationships cause confusion.

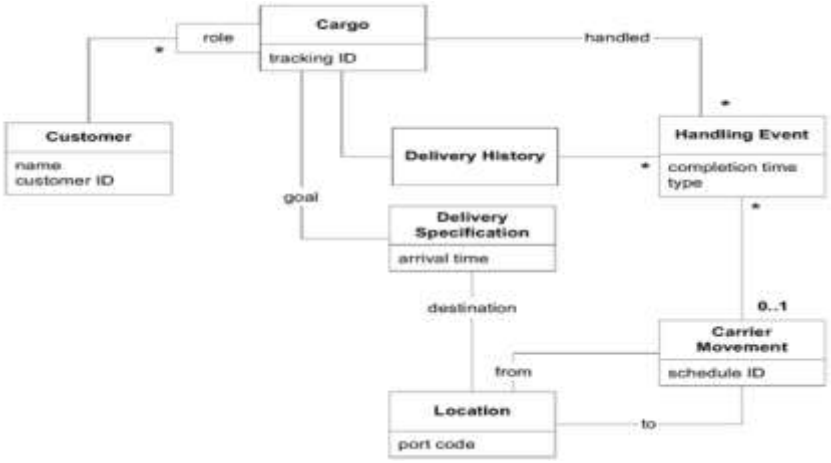
- **Refactoring and Data Migration**
- The tradition of refactoring that is common in the object world has not greatly affected relational database design, because of data migration issues.
- If the object model and the database model diverge, transparency can be lost.

General Principles

- **Simple Directness**: For relational databases acting as the persistent form of an object-oriented domain, simple directness is best. A table row should contain an object and a foreign key should translate to a reference to another **ENTITY** object.
- **Transparency**: Mappings should be transparent and easily understandable by inspecting the code or reading entries in the mapping tool.
- **Avoid Multiple Models**: The arguments for avoiding separate analysis and design models also apply to this mismatch between database and object models.
- **Accept Limitations**: It’s sometimes worth accepting model limitations to keep the mapping very simple.
- **Compromise Where Needed**: Be prepared to make compromises in the database design, such as selective denormalisation, to simplify the object mapping, but be aware of the impact of these trade-offs.

- **Maintain Cohesion**: Even though deviations are sometimes necessary, do not abandon the principle of simple mappings altogether.

Chapter 7: Using the Language: An Extended Example



Model-driven design for a cargo shipping system.

Problem Statement The initial requirements of the cargo shipping system include three basic functions:T

1. Tracking key handling of customer cargo
2. Booking cargo in advance
3. Sending invoices to customers automatically when the cargo reaches some point in its handling

Initial Model

The initial model is based on key concepts like **Cargo**, **Customer**, **Handling Event**, **Delivery Specification**, **Delivery History, Carrier Movement,** and **Role**.

Each of these concepts has a clear meaning within the shipping domain, and provides a basis for the team's **UBIQUITOUS LANGUAGE**.

Design Considerations and Trade-offs

Value Objects vs. Entities: Identify Role as a **VALUE OBJECT** because it has no history or continuity and can be shared. Other attributes like timestamps and names are also considered **VALUE OBJECTS**4.

Associations: Refining associations between objects, constraining traversal directions to reflect the business needs. For instance, the association from Handling Event to Carrier Movement is made traversable, but not the reverse. This decision reflects the business need to track cargo, not ships.

Circular References: Circular reference between Cargo and Delivery History via Handling Events, lead to change the initially consideration from implementing Delivery History as a List object to replacing it with a database lookup for performance and maintainability. **Implementation choices help to avoid holding the same information in two places that must be kept synchronized.**

Repositories: The team identifies five **ENTITIES** that are roots of **AGGREGATES**: Customer, Location, Carrier Movement, and Cargo, each of which is a candidate for a **REPOSITORY**. The choice of which entities should actually have a **REPOSITORY** is based on the application requirements. For example, because users need to select customers, locations, and carrier movements, the system needs Customer, Location and Carrier Movement repositories. A Cargo Repository is required to look up which cargo has been loaded. A Handling Event Repository was later decided as a database lookup was desired to unclutter the implementation – as it was no longer stored within Delivery History.

Aggregate Boundaries: The team uses **AGGREGATES** to define transactional boundaries within the model, with Cargo, Customer, Location and Carrier Movement each being roots of their respective **AGGREGATES**. Handling Event is also an AGGREGATE root.

Modules: The modules should be based on broad domain concepts, not on technical implementations, Based on the given design the modules are "Customer," "Operations," and "Billing".

Key Components and Their Roles

Cargo: Represents the goods being shipped, with associations to Delivery Specification, Delivery History, and Customer, it also contains the history of handling events.

Customer: Represents the customer with different roles for different Cargoes.

Handling Event: Represents a discrete action taken with the cargo, such as loading or unloading.

Delivery Specification: Defines the delivery goal of the Cargo.

Delivery History: Represents what has happened to the cargo, computed from the handling events and carrier movements.

Carrier Movement: Represents the movement of the cargo between two locations using a Carrier.

Role: Represents the different roles the customer plays for a particular cargo.

Repositories: Provides access to AGGREGATE roots and enables the application to persist, find, and retrieve objects.

Modules: Are used to organise the code according to domain concepts.

Allocation Checker: It is implemented as a **SERVICE** and translates between the cargo shipping system's model and the Sales Management System and exposes only what the booking application needs.

Enterprise Segment: A new class added as a **VALUE OBJECT** that is derived for each Cargo and is used to accommodate categories of cargo. It simplifies the interfaces and enriches the domain model.

Integration with the Sales Management System To implement the new requirement to integrate Sales Management System with the cargo shipping system to check cargo allocation. To maintain a clear model-driven design, a new **ANTICORRUPTION LAYER** using an Allocation Checker class was introduced that acts as a **SERVICE** to translate between the models of the two systems and to recast the problem according to the cargo shipping system domain.

Model Refinements Scenario

Dring refinement and refactoring, the concept of an explicit Itinerary is introduced in the context of a cargo shipping application, where it plays a crucial role in linking the booking and operations aspects of the business. Initially, the Itinerary was implicit in the system, with its data scattered across various parts of the booking application, and was primarily used for generating a customer report. However, through conversations with domain experts, it became clear that the Itinerary was a central concept that needed to be explicitly modelled.

The system had all the data needed to create an itinerary, but it was scattered and not represented as a cohesive object3. The Itinerary was implicitly present in a report generated by the booking application, which contained the vessel voyage, loading port, unloading port and dates1. By making the Itinerary an explicit object, the team elevated it from a mere piece of report data to a first-class concept in the domain model.

The domain experts considered the Itinerary a key link between booking and operations, as well as customer relations. This highlighted that the Itinerary was more than just data, it was a crucial component in the domain1. The operations team would rely on the Itinerary to plan handling work, requiring the loading and unloading sequence with corresponding dates. Customers also used the Itinerary for their planning.

The explicit Itinerary represents a shift from a technical view of the data to a domain-centric view, making the system more expressive, maintainable, and aligned with the business needs. It demonstrates the importance of the UBIQUITOUS LANGUAGE and how explicit modeling of domain concepts can lead to a more robust design.

Benefits of an Explicit Itinerary:

- **Improved Routing Service Interface**: The explicit Itinerary allowed for a more expressive interface for the Routing Service. Instead of the Routing Service putting data directly into database tables, it now returned an Itinerary object. This change decoupled the Routing Service from the booking database.
- **Decoupling of Services**: The Routing Service no longer needed to be aware of the booking database tables. It was now responsible for creating the Itinerary which could be saved by the booking application. This promoted a more modular and maintainable system.
- **Clarified Relationships**: It clarified the relationship between the booking and operations support applications, as they both now shared the explicit Itinerary object.
- **Reduced Duplication**: The explicit Itinerary object eliminated duplication of logic for deriving loading/unloading times by the booking report and the operations support application. The logic for generating the report and the logic used by the operations support application was consolidated.

Domain Logic in Domain Layer: The domain logic was removed from the booking report and placed in the domain layer, isolating and encapsulating the domain logic4.

Enhanced Ubiquitous Language: It expanded the team's UBIQUITOUS LANGUAGE, allowing for more precise discussions about the model. A single concept they could refer to when discussing aspects of both the booking and operations applications.

Refactoring: Developers refactored the code to reflect the new model, quickly. This refactoring process demonstrated how the explicit concept could be incorporated into the design, and how it could positively impact the system.

Itinerary Composition: An Itinerary is made up of Leg objects, each of which has the vessel voyage, load and unload locations and time. The Leg can derive the times from the vessel voyage schedule.

The model: The model came to include the following concepts: a Route made of Legs. The business experts, however, saw routes as having five logical segments, rather than just a string of legs, requiring a more complex model to be built.

Enterprise Segment: An Enterprise Segment was added to the domain model as a **VALUE OBJECT**, this was done to reabstract the domain of the sales management system2.

Key Takeaways

Ubiquitous Language: The importance of a shared language that is used across all aspects of the project, from conversations to code.

Model-Driven Design: The model should guide the design and implementation, with design choices based on domain insights.

Iterative Development: The model, design, and implementation should evolve together in an iterative process.

Balancing Simplicity and Performance: Design trade-offs should consider both simplicity of implementation and performance.

Modular Design: Code should be organised using modules that align with domain concepts.

Strategic Integration: When integrating with other systems, use an **ANTICORRUPTION LAYER** to prevent contamination of the domain model.

Explicit Concepts: Make implicit concepts explicit in the model to enhance clarity.

Domain Expertise: It is important for developers to understand the concepts of the domain, and to engage with the domain experts to ensure the model is accurate and relevant.

Declarative Style: Where possible, use a declarative style of design where code reads like a conceptual definition of the business transaction.

Chapter 8: Using the Language: Breakthrough

These breakthroughs are not planned events, but rather, they emerge after periods of continuous learning and refactoring, and they are often triggered by a recognition of a fundamental flaw or an implicit concept.

- **Breakthroughs are emergent**: Deep models often emerge through a series of small refactorings, with insights often coming unexpectedly.
- **Continuous Refactoring**: Continuous refactoring and model refinement are crucial for setting the stage for breakthroughs.
- **Domain Understanding**: Breakthroughs often involve aligning the model with a deeper understanding of the domain.
- **Explicit Concepts**: Making implicit concepts explicit in the model and code can significantly simplify the design.
- **Ubiquitous Language**: A well-defined and shared **UBIQUITOUS LANGUAGE** is crucial for effective communication and a deeper understanding of the domain.
- **Simpler Designs**: Breakthroughs often lead to simpler designs that are easier to understand and maintain.
- **Not an End State**: A deep model isn't necessarily a final destination, it provides a clearer field of vision for future improvements.
- **Value of Change**: The value of software lies in its ability to change and adapt. A deep model and supple design allow for ongoing change.

DEEP MODELS and supple designs are not created at the outset, but emerge through a process of continuous learning, refactoring, and a willingness to challenge initial assumptions. Breakthroughs are not just about fixing problems, but they represent a fundamental shift in understanding that leads to more elegant and effective solutions.

Chapter 9: Making Implicit Concepts Explicit

"Making Implicit Concepts Explicit," focuses on how to identify and incorporate hidden, but important, ideas into a domain model. The deep model includes essential concepts and abstractions that express users' activities, problems, and solutions.

Main Concepts

- **Implicit Concepts**: These are ideas that are necessary to understand the model or design, but are not directly mentioned or represented. They often emerge through discussions, awkward designs, or contradictions.
- **Making Concepts Explicit:** This involves recognising these implicit concepts and representing them in the model using objects or relationships. This process can be a gradual refinement or sometimes a breakthrough leading to a deeper model.

Scenarios

- **The "Item on a Report" Scenario:**
  Scenario: Users refer to a specific item on a report, which is compiled from various attributes and database queries. The same data is assembled in other parts of the application. However, the need for a specific object to represent this concept has not been recognized.
  Explanation: The developers realize that the item's name represents a significant domain concept. They then create a model object to represent this, leading to a deeper understanding and better communication with the domain experts.
  Key takeaway: Listen to the language of the users and pay attention to the terms they use consistently; these terms might indicate an important implicit concept.
- **The "Itinerary" Scenario:**
  Scenario: A shipping expert consistently uses the concept of an "itinerary," but it is not explicitly represented in the model, instead its data and behaviour were embedded in a report.
  Explanation: The developers create an explicit "Itinerary" object in the model. This improves opportunities for handling the data, behaviour, and communication more clearly.
  Key takeaway: Even if data is already collected, creating an explicit object can clarify domain meaning and open opportunities for further refactoring and improvement.
- **The "Interest Due" Scenario:**

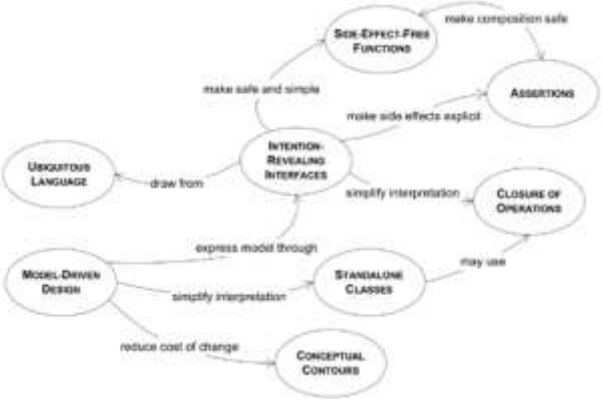
Scenario: A developer tries to track interest due but unpaid within an accounting period. However, the expert explains that interest earned and payment are separate postings.
Explanation: The developers adapt the model to reflect this by modelling payment and interest as separate concepts, rather than trying to combine them. This also introduces the concept of "accrual".
Key takeaway: When inconsistencies or contradictions in statements arise, consider making a distinction between concepts by modelling them separately. This may lead to a deeper understanding of the domain and the software implementation can more accurately reflect the business processes.

- **The "Earning Interest by the Book" Scenario:**
  Scenario: A developer struggles with the complexity of an "Interest Calculator". Instead of consulting a domain expert, she reads an accounting book and finds a whole system of well-defined concepts.
  Explanation: The developer uses the book to understand interest accrual and introduces an "accrual" concept in the model, separating payment from accrual. This also improved her ability to engage with the domain expert.
  Key takeaway: Literature from the domain can provide insights into fundamental concepts.
- **The "Explicit Constraints" Example:**
  Scenario: A "Bucket" object has a capacity constraint.
  Explanation: The invariant that the contents of the bucket must not exceed its capacity is enforced using logic in each operation of the Bucket.
  Key takeaway: Constraints can be explicitly modelled and enforced, improving the robustness of the model.
- **The "Processes as Domain Objects" Example:**
  Scenario: A cargo routing process exists in the domain, and could be implemented using a complex algorithm.
  Explanation: Rather than being treated as a hidden function, the process is made explicit using a SERVICE, while the algorithm itself is treated as a swappable STRATEGY.
  Key takeaway: Domain processes should be made explicit in the model, especially when they are important to the domain experts, while also keeping the complexity of the implementation encapsulated.
- **The "Specification" Example:**
  Scenario: A complex rule needs to be applied when determining if an invoice is delinquent.
  Explanation: The rule is extracted from conditional logic and represented as a "SPECIFICATION" object, which is an object that determines if a certain condition is satisfied. A FACTORY can configure a SPECIFICATION with the required information. This simplifies the main code, makes the rule explicit, and allows it to be configured from various data sources.
  Key takeaway *Complex rules or constraints can be made explicit by using a SPECIFICATION, making it clear what is being checked.*

Key Takeaways

- **Listen to the Language:** Pay attention to the terms used by domain experts; they often indicate important concepts.
- **Scrutinise Awkward Designs:** Look for areas where the design feels complicated or unnatural; these areas might indicate implicit concepts.
- **Address Contradictions:** Investigate contradictions or inconsistencies; they may point to concepts that need to be separated or made explicit.
- **Use Domain Knowledge:** Refer to literature or analysis patterns to help discover concepts that are well-established within the domain.
- **Consider Constraints and Processes:** Beyond simple nouns and verbs, think about constraints and processes as candidates for explicit model concepts.
- **Refactor Iteratively:** Model refinement is an iterative process of knowledge crunching and refactoring, where successive iterations may lead to breakthroughs.
- **Make it Explicit:** Explicitly represent these concepts in the model with objects or relationships to enhance communication and the model's expressive power.

Chapter 10: Supply Design



Pattern that contributes to supple design.

Below listed are the several key concepts aimed at making software more flexible and easier to work with, focusing on how a design can serve both the client developer and the maintenance developer. A supple design complements deep modelling and allows developers to combine elements in predictable ways.

Intention-Revealing Interfaces

- This concept focuses on the importance of naming classes, methods, and other design elements to clearly communicate their purpose and effect, without needing to understand their internal implementation.
- The goal is to make it easy for client developers to use components without getting bogged down in the details of how they work.
- **The names of classes and methods should reflect the concepts they represent and should conform to the UBIQUITOUS LANGUAGE**, enabling team members to quickly infer their meaning.
- The idea is that the interface should reveal the intention of the design, and that all public elements of a design together should make up its interface.
- All tricky mechanisms should be hidden behind abstract interfaces that communicate intentions rather than the means of execution. It advises to describe state relationships and rules, but not how they are enforced; describe events and actions, but not how they are carried out.
- An example of this is when an object encapsulates a rule or calculation, the client code should not have to think of step-by-step software procedures. The object's interface should be expressed in terms of higher-level concepts.
- It advocates for writing tests for behaviour before creating it, to force thinking into client developer mode. TDD approach.

Side-Effect-Free Functions

- This section discusses the distinction between commands and queries. Queries obtain information from the system, whereas commands modify the state of the system.
- **Side effects are defined as any changes in the state of the system that can affect future operations**. The chapter advises to narrow the meaning to any change in the state of the system that will affect future operations.
- Operations that return results without producing side effects are called **functions**.
- **Functions are easier to test and lower risk because they can be called multiple times and return the same value each time**. They also can call on other functions without worrying about nesting.
- The chapter advises keeping commands and queries strictly segregated in different operations5. Methods that cause changes should not return domain data and should be kept simple, while all queries and calculations should be in methods that cause no observable side effects5.
- Another technique mentioned is to create and return a new **VALUE OBJECT** instead of modifying an existing object, as VALUE OBJECTS are immutable5.
- It emphasizes that, where possible, logic should be placed in functions, and commands should be simple, without returning domain information6. Complex logic should be moved into VALUE OBJECTS when a suitable concept presents itself6.

Assertions

- This concept is about making explicit statements about the state of objects and the effects of operations, using preconditions, postconditions and invariants.
- **Postconditions describe the side effects of an operation**, the guaranteed outcome of calling a method.
- **Preconditions are conditions that must be satisfied for the postcondition guarantee to hold**.

- **Class invariants make assertions about the state of an object** at the end of any operation.
- These assertions describe states, not procedures, making them easier to analyse.
- **Assertions simplify a client developer's job by making objects more predictable**, and the effects of delegations should be incorporated into the assertions.
- The chapter advises stating postconditions of operations and invariants of classes and aggregate. If assertions cannot be coded directly, automated unit tests should be written for them, and these can also be written into documentation or diagrams.
- It emphasizes that a design with a coherent set of concepts helps a developer infer the intended assertions, and reduces the risk of contradictory code.

Conceptual Contours, which refers to an underlying consistency within the domain itself. When a model reflects these contours, it can naturally accommodate change. This means that the design should align with the inherent structure and relationships within the problem domain. The goal is to create a design that feels natural and intuitive for domain experts, making it easier to understand and adapt to evolving requirements.

Example: In a loan tracking system, the original design used Calculator classes to work out schedules for different types of fees and interest using conditional logic. Through refactoring, the developer realised that these schedules could be made more explicit by exploding them into discrete classes for different types of fees and interest. The payments of fees and interest, which had been previously separate, were then combined. This resulted in a new model that contained only one more object than the original design, but significantly changed the partitioning of responsibilities. By recognising the cohesiveness of the Accrual Schedule hierarchy, the developer believed that the model now better followed the **Conceptual Contours** of the domain.

Conceptual Contours:

- **Alignment with the Domain**: A design that aligns with the **Conceptual Contours** of the domain feels more natural and easier to work with for domain experts.
- **Improved Adaptability:** When a model reflects the underlying consistency of the domain, it is easier to modify and extend in response to changes in requirements.
- **Natural Accommodation of Change:** Designs that follow the **Conceptual Contours** tend to accommodate changes more naturally because the structure of the software mirrors the structure of the domain.
- **Increased Cohesion:** Aligning with the **Conceptual Contours** often leads to a more cohesive design, where elements fit together in a way that makes sense within the context of the domain.
- **Intuitive Design**: By following these contours, the design becomes more intuitive, revealing the potential of the underlying model and making it easier for a client developer to utilize it. This involves a minimal set of loosely coupled concepts that allow expression of a range of scenarios.
- **Clearer Understanding**: A design based on conceptual contours makes it easier for developers to understand the system, and to anticipate the effect of a change.
- **Deeper Insight**: Designing in this way allows the design to evolve towards deeper insight, which is part of the iterative cycle of refinement

In summary, **Conceptual Contours** is about designing software that closely aligns with the natural structure and relationships of the problem domain, making it more adaptable and easier to understand and modify.

Standalone Classes focuses on the idea of reducing dependencies between different parts of a software system to make it easier to understand, test and maintain.

Key Concepts

- **Interdependencies cause complexity:** The more connections a class has to other classes, the harder it becomes to understand. These connections, or dependencies, can include associations, method arguments, and return values. They force a developer to consider multiple classes and their relationships at once, leading to mental overload.
- **Implicit concepts add to the problem**: Unclear or hidden dependencies make a design even more difficult to understand.
- **Aim for Zero Dependencies**: The goal is to create classes that can be understood entirely on their own, without needing to know about other classes.

- **Standalone classes are easier to handle**: A class with zero dependencies can be studied and tested independently, easing the overall cognitive load of understanding the whole system.
- **Not all dependencies are bad**: Dependencies on basic building blocks like integers or standard library classes don't usually add to the intellectual load. The focus is on removing non-essential dependencies.
- **Factoring**: The process of refactoring code to achieve standalone classes starts with the model itself, and continues with individual associations and operations.
- **Standalone classes within modules**: It is more acceptable to have dependencies between classes within the same module rather than across modules.

The Paint Mixing Scenario:

Initial problem: In the initial paint mixing example, the Paint object directly held three integers representing red, yellow and blue. This created an implicit dependency on the concept of color and how it was represented.

The solution: The creation of the PigmentColor object didn't increase the number of concepts or dependencies. It made the existing concepts more explicit and easier to understand.

Standalone PigmentColor: The PigmentColor class was designed as a standalone class. The concept of color, even of pigment, can be considered separately from paint. By making the two concepts distinct, and by distilling their relationship, the one-way association became more meaningful. The PigmentColor class, where the computational complexity lay, could be studied and tested in isolation.

Benefit of refactoring Paint Class

- **Mental Overload**: The core problem is that dependencies make it harder for developers to grasp the overall design2. Too many connections lead to 'mental overload'.
- **Testing and Maintenance**: Complex dependencies also make testing and maintenance difficult.
- **Implicit vs Explicit**: The section emphasizes making implicit concepts explicit. For example, instead of using three integers to represent a colour, create a class called Pigment Color.
- **Low Coupling**: The section promotes low coupling as a fundamental part of good object design and aims to push this to its extreme by striving for standalone classes.
- **Focus on essential dependencies**: The aim is not to eliminate all dependencies but to remove non-essential ones. It suggests that every dependency should be questioned.
- **Balance**: The concept of paint is fundamentally linked to colour, but colour can be considered without paint. By clearly separating these two concepts the single association between them becomes meaningful.
- **Refactoring** The process of removing non-essential dependencies will help to refine the model and provide a clearer definition of the class.
- **Standalone classes aid understanding**: Standalone classes allow the developer to better understand the module as each class can be analysed in isolation.

In summary, the "**Standalone Classes**" section advocates for designing classes that are as independent as possible to reduce complexity, improve understanding, and enhance maintainability. The paint example shows how making concepts more explicit and reducing dependencies can lead to a more supple design.

Closure of Operations focuses on a design principle that enhances the predictability and composability of operations, especially within **Value Objects**. The core idea is that an operation should return a result of the same type as its input, or the type of the object on which it is operating, which simplifies how those operations can be combined. Here's a breakdown of the key concepts, the examples provided, and the scenarios described:

Key Concepts

- **Closed Operation:** A closed operation is one where the return type is the same as the type of its arguments or the type of the object on which it is operating. For example, multiplying two real numbers always results in another real number, demonstrating that real numbers are "closed under the operation of multiplication".

- **Simplified Interpretation:** Closure makes it easier to interpret an operation and its effect. It reduces complexity by ensuring that the result of an operation can be used as an input to the same operation without unexpected type changes. This makes it easier to chain or combine closed operations.
- **High-Level Interface:** Closed operations allow a high-level interface without introducing dependencies on other concepts.
- **Most Applicable to Value Objects:** This pattern is most often applied to operations of a **Value Object** because the life cycle of an **Entity** has significance in the domain, meaning a new one cannot just be created. In contrast, **Value Objects** are immutable and are often created in response to a query.
- **Reduced Dependency:** Using a closed operation reduces the reliance on other concepts. By ensuring the input and output are of the same type, there is less need to be aware of other types in the system.
- **Immutability:** The pattern of a closed operation often works well with immutable objects. In the case of a **Value Object**, being immutable implies that apart from initializers, all operations are functions. This implies there are no side effects, which improves the reliability of the design.
- **Flexibility:** Operations can be closed under an abstract type allowing concrete classes to be different.
- **Partial Closure:** Operations that are not completely closed may still offer some advantages. This occurs when the return type is a primitive or library class, as this adds little to the cognitive load.

In summary, the "Closure of Operations" section promotes designing operations that maintain type consistency. This results in code that is simpler to understand, easier to compose and test, and has reduced dependencies.

Declarative Design discusses the concept of writing code that expresses the what rather than the how, moving towards a more specification-like approach where the desired outcome is declared, and the system figures out the implementation details. This approach aims to reduce boilerplate code, making the system easier to understand, maintain, and less prone to errors.

Key Concepts

- Executable Specification: Declarative design treats code as an executable specification, where the program's logic is described in terms of its properties and outcomes rather than a series of step-by-step instructions.
- Abstraction and Encapsulation: This approach builds on the principles of abstraction and encapsulation to hide the complex implementation details, which allows developers to focus on higher-level concepts.
- Reduced Boilerplate: By specifying the desired result, much of the repetitive "boilerplate" code needed to achieve that result can be avoided, reducing the risk of error and improving readability.
- Formal Rigour: Declarative design aims to bring a formal rigour to programs that is not always possible with traditional object-oriented programs.
- Clarity of Intent: Declarative code clearly expresses the intent of the program, making it easier for developers to understand the code's purpose and behaviour.
- Moving Towards Declarative: The chapter explains that having Intention-Revealing Interfaces, Side-Effect-Free Functions, and Assertions are steps towards a declarative style. The more of these a design has, the more declaritive it can be.
- Not a Holy Grail: While beneficial, declarative design isn't a perfect solution. It can introduce its own complexities and challenges, such as needing to understand the underlying mechanisms and limitations of the declarative tool.

Declarative Style of Design explores how to move towards writing code that focuses on what the program should achieve, rather than how it should do it. This approach aims to create more understandable, maintainable, and less error-prone software, effectively turning code into an executable specification. It's a step beyond traditional object-oriented programming, emphasizing the expression of intent and desired outcomes.

Key Concepts

- Focus on 'What', Not 'How': Declarative design shifts the emphasis from procedural steps to describing the desired result or state. It allows developers to express the program's logic in terms of its properties and outcomes instead of step-by-step instructions.
- Executable Specifications: The goal is to make the code read like a specification of the problem being solved, rather than a set of instructions. This means the code should clearly state what is to be achieved.

- Abstraction and Encapsulation: This style relies on abstraction and encapsulation to hide implementation details, allowing developers to focus on the higher-level logic.
- Reduced Boilerplate: Declarative design aims to minimize the amount of repetitive code needed to achieve a task. By specifying the result, you can often avoid writing many lines of code that would otherwise be needed1.
- Formal Rigour: Declarative design aims to bring a formal rigour to programs that is not always possible with traditional object-oriented programs.
- Clarity of Intent: Declarative code should clearly communicate the purpose of the code, making it easier to understand and maintain.
- Building Towards Declarative Design: The chapter notes that using Intention-Revealing Interfaces, Side-Effect-Free Functions, and Assertions are steps towards achieving a more declarative style. The more of these a design has, the more declaritive it can be.
- Not a Perfect Solution: Although it has benefits, declarative design has its own challenges and it may not be the best choice for every scenario. There are limitations and pitfalls to this approach.

The idea of the declarative design is to focus on what needs to be achieved instead of how it needs to be achieved.

1. **Combining Specifications with Logical Operators:**

```
Specification ventilatedType1 = new ContainerSpecification(VENTILATED_TYPE_1);
Specification ventilatedType2 = new ContainerSpecification(VENTILATED_TYPE_2);

Specification both = ventilatedType1.and(ventilatedType2);

Specification either = ventilatedType1.or(ventilatedType2);

Specification nither = (ventilated.not()).and(armored.not());


---Implementation
public interface Specification {
boolean isSatisfiedBy(Object candidate);

Specification and(Specification other);
Specification or(Specification other);
Specification not();
}

---
public class ContainerSpecification implements Specification {

private ContainerFeature requiredFeature;

public ContainerSpecification(ContainerFeature required) {
requiredFeature = required;
}

boolean isSatisfiedBy(Object candidate){
if (!candidate instanceof Container) return false;
return
(Container)aContainer.getFeatures().contains(requiredFeature);
}
}

public class NotSpecification extends AbstractSpecification {
    Specification wrapped;

    public NotSpecification(Specification x) {
       wrapped = x;
    }
    public boolean isSatisfiedBy(Object candidate) {
       return !wrapped.isSatisfiedBy(candidate);
    }
}
```

Note:

1. The code focuses on *what* the specifications are, rather than *how* they are evaluated. The developer declares the rules rather than writing the logic to test each of them explicitly, thus making the code more expressive.

2. The use of logical operators makes it clear what the intention of the code is by using the names of the operators themselves, without having to know the implementation details.

2. **Intention-Revealing Interfaces Example :**

```
Map distribution = aLoan.calculatePrincipalPaymentShares(paymentAmount);
aLoan.applyPrincipalPaymentShares(distribution);
```

NOTE : The code uses **Intention-Revealing Interfaces** that express the intent of the operations, not how they are carried out. This allows the client code to focus on the overall transaction.

Angles of Attack

Angles of Attack discusses how to approach the challenge of making a complex software system more supple. It acknowledges that it's unrealistic to try to apply supple design principles to an entire system at once. Instead, it suggests focusing on specific areas and provides a couple of strategies for choosing these targets. One **need to be strategic about where you apply these design principles**, picking specific parts of the system to focus on to get the most impact.

- **Targeted Approach:** You can't make an entire large system supple all at once. Instead, focus on specific parts.
- **Strategic Refactoring:** Choose areas where applying supple design will have the most significant positive effect. This approach to refactoring makes the process more manageable and impactful.
- **Iterative Improvement:** The process of making a system supple is iterative. You work on one area, improve it, and then move to another area.
- **Prioritization**: You need to prioritise what parts of the system you are going to work on. Some areas will yield more benefit than others.

Strategies: There are two main strategies for selecting which parts of the system to focus.

1. **Carve Off Subdomains:**
   - **Identify Specialised Areas**: Look for parts of the system that have a specific, well-defined purpose or that correspond to a specialised area of the business.
   - **Separate and Refactor**: Separate these areas into their own modules, and then apply supple design principles to make those modules cleaner, more understandable and robust. This approach helps to reduce the overall complexity of the system.
   - **Declarative Style**: After separating a subdomain, the code left behind can often be written in a more declarative style, using the newly created module.
   - **Incremental Improvement**: By carving out subdomains, you work towards making the overall design more supple by breaking the larger problem into smaller, more manageable parts.
   - **Focus on Impact**: It's better to make a big impact on one specific area rather than spreading your efforts too thinly across the whole system.
2. **Address Complex Rules or Logic:**
   - **Identify Bottlenecks**: Look for parts of the system that are difficult to understand, change or extend. This approach makes sure you are addressing a genuine problem.
   - **Pull out Rules and Logic**: Extract complex rules or logic into a separate model or a simple framework. This moves the rules out of the normal application flow.
   - **Declarative Expression**: This kind of refactoring makes it possible to express the extracted logic or rules in a declarative style that is more readable and maintainable.
   - **Clearer Application Core**: This approach reduces the complexity of the core application code and separates it from specialised logic.

The "Shares Math" Problem

The "Shares Math" problem is based on a scenario in a syndicated loan system. In this system, a loan is often funded by multiple lenders, each with a specific share of the loan. When a borrower makes a payment, that payment needs to be distributed to the lenders proportionally to their share. This involves complex calculations and logic.

Initial Code (Before Refactoring)

The initial code is not explicitly provided in the sources, but the text implies that it likely involves procedural code with complex calculations within the Loan class. The sources suggest that this initial code was:

- Difficult to understand.
- Hard to extend with new functionality.
- Not clearly related to the underlying business concepts.

The below code shows how the Loan object was responsible for calculating how to distribute a payment across all the lenders. This meant that the Loan object was not just responsible for the loan, but also the logic of how a payment should be distributed.

```
public class Loan {
    private Map<Lender, Double> lenderShares;
    private double totalLoanAmount;

    // ... other loan attributes and methods ...

    public Map<Lender, Double> calculatePrincipalPaymentShares(double paymentAmount) {
        Map<Lender, Double> distribution = new HashMap<>();
        for (Map.Entry<Lender, Double> entry : lenderShares.entrySet()) {
            Lender lender = entry.getKey();
            double share = entry.getValue();
            double amount = (share / totalLoanAmount) * paymentAmount; // Complex calculation
            distribution.put(lender, amount);
        }
        return distribution;
    }

     public void applyPrincipalPaymentShares(Map<Lender, Double> distribution) {
        // Logic to apply payment to each lender based on distribution
        // ...
    }
}
```

Problems with the above code

- **Complex Logic**: The code mixes the core state of a Loan object with the logic for calculating and distributing payments, making the Loan class more complex.
- **Low-Level Operations**: The code deals with individual shares and low-level calculations directly, obscuring the underlying concept of how the shares relate to each other as parts of a whole.
- **Difficult to extend**: Adding new payment types or rules for distribution would require further modifications to the Loan object and would make the code even more complicated and hard to understand.

Refactored Code

The refactored code introduces a *SharePie* object, which encapsulates the logic of how to distribute payments, as a **VALUE OBJECT.**

```
public class SharePie {
    private Map<Lender, Double> shares = new HashMap<>();

    public SharePie(Map<Lender, Double> shares) {
        this.shares = shares;
    }

    // Accessors and other straightforward methods are omitted

    public double getAmount() {
        double total = 0.0;
        Iterator<Lender> it = shares.keySet().iterator();
        while (it.hasNext()) {
            Lender lender = it.next();
            total += shares.get(lender);
        }
        return total;
    }

    public SharePie prorated(double amount) {
          Map<Lender, Double> newShares = new HashMap<>();
          double total = getAmount();
          for(Map.Entry<Lender, Double> entry : shares.entrySet()){
             Lender lender = entry.getKey();
```

```
            double share = entry.getValue();
            double newAmount = (share / total) * amount; // Calculation is now in SharePie
            newShares.put(lender,newAmount);
        }
        return new SharePie(newShares);
    }


    public SharePie plus(SharePie other) {
       Map<Lender, Double> newShares = new HashMap<>(this.shares);
       for (Map.Entry<Lender, Double> entry : other.shares.entrySet()) {
            Lender lender = entry.getKey();
            double amount = entry.getValue();
            newShares.put(lender, newShares.getOrDefault(lender, 0.0) + amount);
       }

        return new SharePie(newShares);
    }

    public SharePie minus(SharePie other) {
        Map<Lender, Double> newShares = new HashMap<>(this.shares);
        for (Map.Entry<Lender, Double> entry : other.shares.entrySet()) {
           Lender lender = entry.getKey();
           double amount = entry.getValue();
            newShares.put(lender, newShares.getOrDefault(lender, 0.0) - amount);
        }
        return new SharePie(newShares);
   }
}

public class Loan {
    private SharePie shares;
    // ... other loan attributes and methods ...

     public void applyPrincipalPayment(double paymentAmount) {
       SharePie paymentShares = shares.prorated(paymentAmount);
        //Logic to apply the share calculations to the loan
        setShares(shares.minus(paymentShares));
    }
}
```

The above example is an illustrates both strategies of "Angles of Attack":

1. **Carving Off Subdomains:** The distribution of payments (the "Shares Math" problem) is a specific subdomain within the larger loan system. The SharePie class can be considered a module that encapsulates these concepts.
    - By creating SharePie, the complex calculations and rules related to share distribution are extracted out of the Loan object. This makes the Loan object more focused and easier to understand.
    - The Loan object does not need to know how the SharePie object works, it just tells it what to do.
    - The SharePie class also can be reused to implement other scenarios where you are splitting or combining resources, which is not directly related to the loan itself.

2. **Addressing Complex Rules or Logic:**
    - The initial way of handling loan shares involved complex, procedural logic that was mixed into the main business logic of the Loan object.
    - By creating the SharePie class, the logic for sharing and distributing payments is pulled out into a separate model, and is now expressed using simple mathematical operations.
    - The SharePie class uses a **declarative style**. For example, the plus() and minus() methods of SharePie return a new SharePie object rather than changing the state of existing SharePie objects. This means that code using SharePie can be more easily understood by looking at its effects, rather than the steps involved.
    - The code in the Loan object becomes very simple, and is now a declarative statement of the effect of the principal payment.

Benefits of the Refactored Code

- **Increased Cohesion**: The SharePie class is now responsible for share calculations, which is more cohesive than before, when the Loan object was responsible for both.

- **Improved Understandability**: The code is now more straightforward to read, as the complex calculations are encapsulated within the SharePie object, behind an **INTENTION-REVEALING INTERFACE** with methods that are named according to what they do.
- **Enhanced Reusability**: The SharePie class can be reused in different parts of the application where distribution calculations are required.
- **Easier to Extend**: New variations of payment distribution can be easily created, as the logic is encapsulated in the SharePie and Share class.
- **Declarative Style**: The use of **SIDE-EFFECT FREE FUNCTIONS** in the SharePie class makes the code more like a conceptual definition of the business transaction. The code in Loan is simplified and is now a statement of how a principal payment changes the state of the Loan.

Chapter 11: Apply Analysis Pattern

Example 1: Interest Calculator

The Problem Statement: The original system had an "*Interest Calculator*" that was becoming increasingly complex. *It was difficult to add new types of interest or handle situations where payments were not made on schedule. The system just kept track of the interest due, rather than the interest earned and when payments were made.* The developers realised that they were using a single "*account*" to keep track of different things, which made the code hard to reason about.

The Solution (Using Analysis Patterns):
A basic accounting model that used **"Accounts"** and **"Entries"**.

- **Accounts** hold a value (like a bank account).
- **Entries** represent changes to that value (like deposits or withdrawals)4.

In the analysis patterns, money isn't just moved into or out of an account, it is also moved from one account to another, using the concept of a **Transaction**.

The initial idea was that an **Entry** would be made into the Interest Account for interest earned, and then another Entry would be made when the payment was made. This would have kept a history of all interest accruals, as well as payment history. However, there was some *confusion about the meaning of a **Transaction**, and whether payments should be tied to accruals, since payment and accruals are separate postings.*

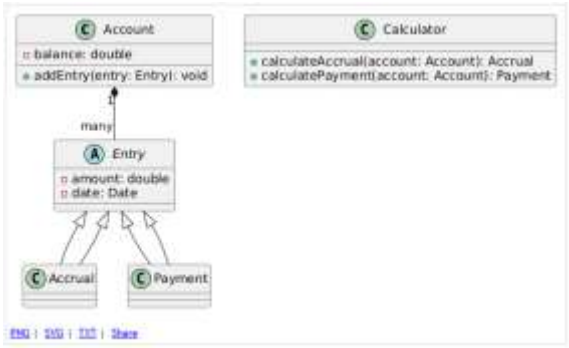
The developers then decided to separate the accrual from the payment, and use **Accounts** to track interest accruals, and also track payments.

The developers ended up with a model where **Entries** were subclassed into **"Payment"** and **"Accrual"** because they had different responsibilities, and because these were both important concepts in the domain.

The Result:

- The new model was easier to understand and test because the most complicated logic was done using **SIDE-EFFECT FREE FUNCTIONS**.
- The core logic was still a **Calculator**, but instead of being tightly coupled to the **Account** itself, the **Calculator** just created entries that were added to the **Account**. This made the **Calculator** simpler and easier to test, as the logic was now purely about calculating an **Entry**, rather than updating an account.
- The code became more explicit, for example introducing the term "accruals" into the model, which was also used by the domain expert.
- The developers also found they had to compromise the abstraction, as the **Entries** for fees and interest had to be stored in different tables in the database because of project standards.

  In summary, the use of the "Account" and "Entry" patterns helped the developers simplify their interest calculations and align their model with established accounting principles.



Example 2: Posting Rules (Nightly Batch Run)

The Problem Statement: The nightly batch process was becoming a dumping ground for complex logic. It was not designed according to any domain principles and was becoming increasingly difficult to maintain.

The Solution (Using Analysis Patterns):

- **Posting Rules** define how entries in one account can trigger entries in another account, which is a core idea in accounting software.
- The **Posting Rule** is triggered by a new entry in its "input" account and then creates a new entry in its "output" account based on a defined calculation.
- **Posting Rules** can be triggered in different ways, for example "**eager firing**" where rules are executed immediately, or a "**posting-rule-based firing**" where a rule is triggered by an external agent, like the nightly batch15.
- They realised that the existing batch was not a set of simple procedures, but was actually implementing a series of **Posting Rules**, so they started to sketch out how the nightly batch would work using the concept of a **Posting Rule**.
- They realised that the batch was tightly coupled with the calculation methods, and that this tight coupling would cause problems in the future.

- The team had some debate about how they were using some of the terminology from the analysis patterns. For example, they had a **Method** that calculated the amount of the posting, but since the amount was simply the full amount being posted, the **Method** wasn't really required16. Also, the **Posting Rule** was responsible for knowing the "ledger name", so the **Method** was not required16.
- They also found they had to make some compromises, as the **Posting Rules** were not linked directly to the **Accounts**. The **Asset** object was used to look up the **Posting Rules** using a **SINGLETON**.
- They were uncomfortable with this setup as the **Asset** object was now involved in the selection of a posting rule, rather than just generating accruals.

The Result:
The refactored code moved the complexity out of the nightly batch into the domain layer. The nightly batch code simply iterated through the assets, sending self-explanatory messages.

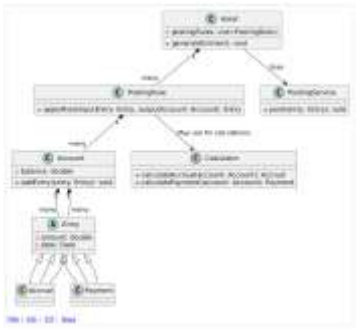
- The **Posting Rules** became objects which encapsulated the logic for updating the accounts using the **Posting Service**, which was a **FACADE** exposing the accounting application's API.
- The nightly batch became simpler, as the domain model was responsible for executing the rules rather than the batch itself.

In summary, by applying the "Posting Rule" pattern, the team was able to make the batch process more explicit, understandable, and maintainable, and made it clear that the batch was not just a set of procedures, but was actually implementing a series of Posting Rules.

Key Takeaways from Both Examples

- **Analysis patterns provide a starting point:** These patterns are not "out-of-the-box" solutions. They offer a structured way to think about common problems and guide the design process.

- **Domain knowledge is crucial:** The developers had to discuss the patterns with their domain experts to understand how they applied to their specific situation. This helped them to identify implicit concepts and how these concepts related to their domain.
- **Iteration and compromise are necessary:** The initial models were refined based on feedback from the domain experts, and developers had to make pragmatic decisions that sacrificed some abstraction in favor of practical concerns such as the project's database structure.
- **Patterns can clarify hidden logic**: Analysis patterns helped developers discover that the seemingly simple nightly batch was actually implementing domain rules that should be made explicit.
- **Model-Driven Design**: The examples show how starting with a model based on an understanding of the domain, rather than starting with code, can lead to software that more closely reflects how the business works.



Analysis patterns are like a collection of best practices for modeling common business concepts. Instead of starting from scratch, developers can draw on these patterns, which have been developed and refined through experience. This approach can save time and effort by avoiding common mistakes and providing a solid foundation for further refinement.

- **Not Out-of-the-Box Solutions** Analysis patterns are not meant to be used as ready-made solutions but rather as a jump start in your project. They are designed to be adapted to specific circumstances13.
- **Provides Cleanly Abstracted Vocabulary** They give you a way to discuss and understand the common concepts of your project.

Chapter 12: Relating Design Patterns to the Model

The chapter focus on how some technical design patterns can be applied within a domain model to better express domain concepts. The chapter primarily focuses on two specific patterns : **STRATEGY** (also known as **POLICY**) and **COMPOSITE.**

STRATEGY (A.K.A. POLICY)

The **STRATEGY** pattern involves **encapsulating different algorithms or rules** within separate objects, making them **interchangeable**. This allows the core logic to vary independently of the client that uses it. In domain-driven design, **STRATEGY** is not just about technical flexibility but also about **representing meaningful domain concepts**, such as policies or business rules.

- **Key Idea:** The core concept is to **separate the varying part of a process from the stable part**. This makes the main process clearer and the different options more visible.
- **How It's Used:** A process or rule that has multiple variations is **factored into a separate "strategy" object**. The core process then uses a specific strategy object to perform its task

Example: Route-Finding Policies

Route-finding policies example shows how **STRATEGY** can be used in a shipping application to determine the best route for a cargo.

A *Routing Service* needs to find a route based on different criteria, such as the fastest or the cheapest route. Without **STRATEGY**, the *Routing Service* would have to have conditional logic throughout its code to handle each case.

Applying STRATEGY design Pattern:

The different criteria (fastest, cheapest, etc.) are represented as separate strategy objects called *Leg Magnitude Policies*.

The *Routing Service* now takes a *Leg Magnitude Policy* as a parameter. It uses this policy to calculate a "magnitude" for each leg of the journey. The service then chooses the route with the minimum total magnitude. The specific strategy for choosing the best route can be changed by passing different *Leg Magnitude Policy* objects to the service. This makes the *Routing Service* flexible and easier to extend with new criteria.

COMPOSITE

The **COMPOSITE** pattern allows you to **treat individual objects and compositions of objects uniformly**. This is useful for representing **hierarchical structures** where you want to apply the same operations to individual parts and whole assemblies. It allows you to define an abstract type that encompasses all members of a **COMPOSITE**.

Key Idea: Clients can treat individual "leaf" nodes and container nodes the same way.

How It's Used: The methods that return information can be implemented on container objects to return aggregated information about their contents. Leaf nodes implement those methods based on their own values.

Example: Shipping Routes

COMPOSITE design pattern can be applied to represent a shipping route.

Initial Situation: A shipping route can consist of various segments, such as standard shipping routes or "door legs," which use local transport. These segments are planned at different times and may have different characteristics.

Applying COMPOSITE design pattern: The *Route* is defined as an abstract type, which can be a *Route segment*, a *door leg*, or a complete route.

Each of these can be treated as a *Route*, allowing for arbitrary nesting and composition of routes. The COMPOSITE pattern simplifies how operations, such as calculating route distance or cost, are performed, as they can be applied uniformly across all types of routes and route segments. This way, the client code can calculate values for part of the route or a complex route using the same operations.

Why the **FLYWEIGHT pattern** is not considered a domain pattern, unlike **COMPOSITE** and **STRATEGY ?**

FLYWEIGHT design pattern does **not** describe any conceptual element of the domain. The **FLYWEIGHT** pattern is purely an implementation detail and has no direct representation within the domain model itself.

The **FLYWEIGHT pattern** is primarily an **implementation technique** used to **optimise memory usage** when dealing with a large number of similar objects. It achieves this by **sharing common data** between objects, using immutable objects to represent shared state. The pattern can be used when implementing **VALUE OBJECTS**, where multiple instances of an object may have the same state. Since Route are conceptual objects are composed of other conceptual objects, hence it needs to be represented with Composite or **STRATEGY** pattern.

Chapter 13: Refactoring towards deeper Insight

"Refactoring Toward Deeper Insight", don't just make the code tidier, but about achieving a deeper understanding of the problem the software is trying to solve. This deeper understanding leads to a more flexible and useful model and ultimately to better software.

- **It's not just about code:** Refactoring isn't just about tweaking code to be more efficient. It's also about improving the domain model, which is the way the software represents the real-world concepts and rules it's dealing with. Sometimes, the code might be fine, but the underlying model is flawed.
- **Look for problems**: Refactoring towards a deeper insight can start in different ways. It might begin when developers notice a problem with the code, or when the language they use to describe the model doesn't match with the domain experts' language. New requirements that don't fit naturally can also trigger a refactor. The key is to recognise when the model is not good enough.

- **Dig deeper**: To improve the model, developers should listen closely to the language used by the team and domain experts, looking for clues about implicit concepts. They should also examine the existing design for awkwardness or contradictions. This often involves conversations with domain experts, and studying relevant literature.
- **Make the implicit explicit**: A key part of this process is recognising implicit concepts and making them explicit in the model using objects or relationships. For example, if the team talks about a "delivery schedule" but it's not represented as an object, making it explicit might provide valuable insight.
- **Continuous Process**: Refactoring toward a deeper insight is not a one-off task. It's an ongoing process of learning, refining and adapting as the team gains more understanding of the domain. The idea is that implicit concepts become explicit, the design becomes more flexible (supple) and can lead to a breakthrough and deeper model.
- **Design for Developers**: The software isn't just for the end-users; it's also for developers. The goal is to create a design that is easy to work with and modify, making it easier to integrate and change the code. A supple design makes it easy to predict the effects of running the code and therefore the effects of any changes.
- **Supple design**: Supple design is a key component of this refactoring process. It makes it easier to anticipate the effects of running the code and the consequences of changing it. Supple design also helps limit mental overload, by reducing dependencies and side effects, and is based on a deep model of the domain that is fine-grained only where it matters to the users.
- **Breakthroughs**: Sometimes, these refinements can lead to a sudden breakthrough in understanding, resulting in a much better model. This is a shift in thinking that requires major design changes. But the possibility usually comes after a number of smaller refactorings.
- **Don't be afraid to change**: When a new understanding of the domain or a new requirement comes along, it might force changes to the model. This is an opportunity to make the model and design even better, as well as make it more supple.

Cultivating a mindset of continuous learning and improvement, where developers and domain experts work together to create a software that truly reflects the intricacies of the domain. It's about making the software easier to understand and change, leading to a better product and a more efficient development process.

Chapter 14: Maintaining Model Integrity

A **Bounded Context** defines the range of applicability of a particular model within a software system. It's like drawing a circle around a specific part of your system where a particular model and its associated Ubiquitous Language are valid and consistent. It is a way to manage complexity by recognizing that a single model can't effectively represent every aspect of a large system.

- **Scope of a Model:** A Bounded Context clarifies where a specific model should be applied and where it should not2. This helps to prevent confusion and maintain consistency within that specific area of the system.
- **Unified Language:** Within a Bounded Context, a single model and its Ubiquitous Language are maintained as consistently as possible. This ensures that everyone working within that context shares a common understanding and terminology.
- **Team Organisation:** Bounded Contexts often align with team organisation, where teams working closely together will naturally share a model context. Different teams, or those who don't communicate, will naturally use different contexts.
- **Explicit Boundaries**: Bounded Contexts are explicitly defined in terms of team organization, usage within specific parts of the application, and physical manifestations such as codebases and database schemas. These boundaries allow teams to keep their models pure and focused.
- **Integration:** When different Bounded Contexts need to interact, they require *translation layers*. This translation is explicitly defined and is not part of either model's Bounded Context.
- **Not Modules**: It is important to note that **Bounded Contexts are not the same as Modules**. Modules are used to organise elements within a single model, whereas Bounded Contexts are about separating different models.

- **Context Map:** A Context Map is used to get a global overview of all the Bounded Contexts in a project and their relationships to each other13. It includes the names of each Bounded Context, explicit translation for any communication, and highlights any sharing of information

Continuous Integration, a process aimed at maintaining the integrity of a model within a defined Bounded Context when multiple people are working on the same project. It involves the frequent merging of all work within the context to ensure consistency and to quickly identify and fix any issues.

- **Frequent Merging:** All code and other implementation components are merged together often. This helps to prevent large divergences between team members' work.
- **Early Detection of Problems:** By merging work frequently, any inconsistencies or conflicts are found quickly, rather than waiting until the end of the project. These inconsistencies are called "*splinters*" and can be between model concepts or the implementation.
- **Automated Tests:** Automated tests are used to identify problems early in the integration process. These tests help to quickly expose any issues created by the merging of code.
- **Shared Understanding:** Continuous Integration also includes constant communication among team members. This helps to cultivate a shared understanding of the evolving model through use of a Ubiquitous Language.
- **Integration of Concepts and Implementation:** Continuous integration occurs at two levels. There is the integration of the model concepts and the integration of the implementation. The Ubiquitous Language is how the team integrates concepts, and systematic merge/build/test processes integrate implementation.
- **Focus on Bounded Context**: Continuous Integration is most effective when applied within a specific Bounded Context, which is a defined scope where a particular model applies and is kept consistent. *This means that the need to integrate with other systems with a different model does not have to be done at the same rate as integration with in the bounded context.*

Context Map is a high-level view of all the different models within a software project and how they relate to each other. It is a tool to help manage complexity when a project has multiple models.

- **Identifying Models**: The first step in creating a Context Map is to identify all the different models being used in a project. This includes both explicit models (like object models) and implicit models (the way different parts of the system are designed). It also means identifying the models used by other teams or other systems that interact with your system.
- **Bounded Contexts**: Each of these models exists within a Bounded Context, which is a defined area where a specific model is applicable and consistent. A context map shows these areas and names them, so they can be discussed.
- **Relationships between Contexts**: The Context Map illustrates how different Bounded Contexts and their respective models relate to each other. These relationships can take different forms, such as:
    - **Shared Kernel**: Where two contexts share a portion of their model.
    - **Customer/Supplier**: Where one context relies on the model of another.
    - **Conformist**: Where one context conforms to the model of another.
    - **Anticorruption Layer**: Where a layer is used to translate between two different models.
    - **Published Language**: Where an agreed-upon language is used for communication.
    - **Separate Ways**: Where two contexts do not interact or share any part of their model.
- **Translation**: When models in different Bounded Contexts need to interact, the Context Map highlights where translation or mapping is needed between them. The Context Map makes it clear where these boundaries are and how they have to communicate.
- **Communication**: The Context Map also provides a shared language for teams to use when discussing different parts of the system, using the names of the Bounded Contexts as part of the project's Ubiquitous Language.
- **Purpose of a Context Map:**

- **Clarity:** The Context Map brings clarity to a project by making all the models and their relationships explicit. This shared understanding is important for effective collaboration and for avoiding misunderstandings about different parts of the system.
- **Boundary Management**: By defining Bounded Contexts, the map helps teams focus on keeping their specific model consistent within its own boundaries. This reduces the risk of different teams corrupting each other's models and implementations.
- **Strategic Decision Making**: Once the existing Bounded Contexts and relationships are visible, it becomes possible to make strategic decisions about which contexts should be unified, which should be separated, and which should be integrated in particular ways. This also makes it possible to identify and manage the interfaces between different models, making the project more cohesive and manageable.
- **Team Alignment**: The boundaries defined by the Context Map tend to follow team boundaries, so a team works within a Bounded Context and is responsible for the model used in that context. Team organization and software models have an important relationship.

The Context Map is a crucial tool for navigating the complexities of software projects by providing a clear picture of all the models, contexts, and relationships involved. It helps to ensure that everyone understands the overall structure of the project and can work together effectively within their own area of responsibility, while also understanding the larger context.

Relationships between BOUNDED CONTEXTS, providing patterns for how different models can interact within a larger system. These patterns help manage complexity and ensure that different parts of a system, which may use different models, can work together effectively.

- **SHARED KERNEL**: This involves two or more teams agreeing to share a common model for a specific part of the system. This shared model, or SHARED KERNEL, is often the core domain or generic subdomains and aims to reduce duplication and make integration easier. The teams working with this shared model must coordinate their changes.
- **CUSTOMER/SUPPLIER**: This pattern describes a situation where one subsystem (the supplier) provides services to another (the customer) with dependencies flowing in one direction. The customer system has little to no influence on the supplier’s model. This is common when different teams or subsystems serve distinct user communities with different models and tools.
- **ANTICORRUPTION LAYER**: This pattern is used when integrating with a legacy or external system that has its own, often incompatible, model. An ANTICORRUPTION LAYER acts as a translation layer, providing an interface to the client in terms of their domain model, thereby preventing the client's model from being corrupted by the external system's model. This layer translates between the two models as needed. It may consist of a FACADE, ADAPTER and Translator.
- **SEPARATE WAYS**: This is when two systems or subsystems operate independently, with no integration or data sharing. This approach simplifies development and reduces the need for coordination, but it also forecloses options for future integration. If integration is needed later, complex translation layers may be required.
- **OPEN HOST SERVICE**: This pattern is used when a subsystem is designed to be easily accessible by multiple other systems. A translation layer is created for each external system to avoid corruption of the models. This approach is suitable for a subsystem that is in high demand.

The relationships between BOUNDED CONTEXTS are influenced by factors like the **level of control**, **the degree of cooperation between teams**, and **the amount of integration needed**. It's important to note that these patterns also serve as a vocabulary to describe existing relationships on a project. The goal is to consciously choose the most suitable relationship type and then organize teams and the software architecture accordingly. Additionally, the choice of relationship strategy impacts the complexity of deployment. For example, a SHARED KERNEL implies more coordination during deployment, while SEPARATE WAYS makes deployment simpler. The relationships between contexts can also be transformed, if required. Common transformations include merging and splitting contexts, or changing the relationships between them. Merging contexts is generally difficult, involving either refactoring one context to match the other or finding a new model capable of assuming the responsibilities of both. The relationships between BOUNDED CONTEXTS can also be used to organise a large-scale structure within an organisation. A large-scale structure

can either exist within a single BOUNDED CONTEXT, or it can cut across multiple contexts and organise the CONTEXT MAP. Testing at the boundaries between contexts is particularly important, helping to compensate for the subtleties of translation and lower levels of communication that may exist there.

CONFORMIST is explained as a specific type of relationship between BOUNDED CONTEXTS. It arises when one team or subsystem decides to strictly adhere to the model of another, upstream team or component, to simplify integration.

- **Definition:** A CONFORMIST relationship means that a downstream team or subsystem adopts the model of an upstream system, thereby eliminating the need for complex translations. This is in contrast to other patterns like an ANTICORRUPTION LAYER, where a translation layer is built to isolate the two different models.
- **Motivation:** The primary driver for adopting a CONFORMIST approach is to reduce the complexity of integration. By aligning with the upstream system's model, the downstream system avoids the challenges of mapping between two different sets of concepts.
- **When to use it**: This pattern is typically used when:
    - Integrating with an off-the-shelf component that has a large interface.
    - The upstream component has a well-defined and robust model with significant knowledge.
    - The downstream system is primarily an extension of an existing system, and the interface between the two is large.
- Implications of using CONFORMIST relationship
    - **Reduced Translation Complexity**: The most significant advantage is the elimination of complex translation logic, as both systems now share a common model.
    - **Shared Language**: It fosters a shared UBIQUITOUS LANGUAGE with the supplier team, which improves communication and reduces misunderstandings.
    - **Limited Design Freedom**: Downstream designers have to work within the constraints of the upstream model. This may prevent the downstream team from implementing the ideal model for their application.
    - **Dependency**: The downstream system becomes heavily dependent on the upstream model and its capabilities. Any changes in the upstream system may have a direct impact on the downstream system.
    - **Extension Only**: The downstream system is limited to adding functionality on top of the existing model without making any modifications.

- **CONFORMIST** Relationship to other patterns:
    - **SHARED KERNEL**: While both CONFORMIST and SHARED KERNEL involve sharing a model, they differ significantly in their development processes. In a SHARED KERNEL, both teams collaborate and coordinate changes. In contrast, the CONFORMIST pattern involves a one-sided relationship, where the downstream team adheres to the upstream model without direct collaboration.
    - **ANTICORRUPTION LAYER**: It is an alternative to using an ANTICORRUPTION LAYER and is chosen when the upstream system is not poorly designed and the team is willing to accept the loss of design independence in return for a simpler integration.

ANTICORRUPTION LAYER is presented as a crucial pattern for managing integration between systems with differing models.

- **Purpose:** An ANTICORRUPTION LAYER acts as an intermediary between a new system and a legacy or external system. Its primary goal is to prevent the new system's domain model from being corrupted by the model of the other system. This is particularly important when the external system has a weak or unsuitable model.
- **Necessity:** This layer becomes necessary when direct integration with another system, especially one with a different or problematic model, threatens to compromise the integrity of the new system's design. Without

it, the new system may start to resemble the external system in an ad-hoc way, leading to a loss of model clarity and maintainability.

- **Key Functions are:**
    - **Translation:** The ANTICORRUPTION LAYER translates between the two models, ensuring that data and actions are interpreted correctly in each system.
    - **Abstraction**: It re-abstracts the other system's behaviour, offering its services in a way that aligns with the client's model. This may involve exposing multiple SERVICES or ENTITIES, each with a specific responsibility in terms of the new system's model.
    - **Isolation**: By providing a distinct interface based on the new system's model, it isolates the new system from the complexities and limitations of the external system.
    - **Implementation**: An ANTICORRUPTION LAYER is typically implemented using a combination of patterns:
        - **FACADES**: A FACADE simplifies access to the external system, hiding its complexity and providing a more streamlined interface. It is written strictly in accordance with the external system's model.
        - **ADAPTERS**: An ADAPTER acts as a wrapper that allows the client to use a different protocol than that understood by the implementer of the behaviour. In this case, it converts messages between the two models.

◦

        - **Translators**: A translator performs the actual conversion of data and objects between the two models. It is a lightweight object instantiated as needed and does not require state.

- Benefits of using an **ANTICORRUPTION LAYER**
    - **Model Integrity**: It protects the new system's model from being distorted by the external system's model.
    - **Flexibility**: It allows the new system to evolve independently of the external system.
    - **Reusability**: The layer can be reused if integration with similar external systems is needed in the future.
    - **Clarity**: The ANTICORRUPTION LAYER provides an explicit and well-defined point of interaction, making the overall system easier to understand and maintain.
- Considerations for **ANTICORRUPTION LAYER**
    - **Complexity**: The ANTICORRUPTION LAYER can become a complex piece of software in its own right.
    - **Bidirectionality**: In some cases, the ANTICORRUPTION LAYER may need to be bidirectional, with SERVICES on both interfaces.
    - **Communication**: Decisions about where to place communication links need to be made pragmatically.
    - **Refactoring**: If you have access to the external system, some refactoring there can simplify the task of creating the ANTICORRUPTION LAYER.
- When to Use **ANTICORRUPTION LAYER**:
    - When integrating with legacy systems or external systems that have different models.
    - When the external system has a large interface.
    - When there is a need to protect a new, well-defined model from the influence of a weaker model in the external system.
- **ANTICORRUPTION LAYER** Relationship to other patterns:
    - **CONFORMIST**: The ANTICORRUPTION LAYER is an alternative to using a CONFORMIST approach. Instead of adhering to the model of the external system, the ANTICORRUPTION LAYER allows the new system to maintain its own model and translates between the two.
    - **OPEN HOST SERVICE**: An ANTICORRUPTION LAYER is created for each external system to avoid corruption of the models when using the OPEN HOST SERVICE pattern

Separate Ways refers to a strategy for handling multiple models within a project where teams or subsystems operate with distinct models and choose to minimise integration. This approach is used when the costs of integrating models outweigh the benefits, often because of differing needs, technologies, or team mindsets.

- **Distinct Models**: Different parts of the system use their own models, which may have different terminologies, concepts, and rules.
- **Limited Integration**: Integration is minimized, with translation layers developed only where absolutely necessary. This is in contrast to approaches like a SHARED KERNEL, where models are more tightly coupled.
- **Independent Evolution**: Models are allowed to evolve independently according to the needs of each context. This autonomy can be beneficial where specific needs require different models.
- **Translation Layers**: When integration is needed, translation layers are created and maintained by the teams involved as the single point of integration. This is in contrast with integration with external systems where the ANTICORRUPTION LAYER typically has to accommodate the other system as is.
- **Team Autonomy**: Typically, there is a correspondence of one team per BOUNDED CONTEXT. One team can maintain multiple BOUNDED CONTEXTS, but it is hard for multiple teams to work on one together.
- **Trade-offs**: Choosing Separate Ways involves a trade-off between seamless integration and the effort required for coordination and communication. It prioritizes independent action over smoother communication.
- **Potential Downsides**: This approach forecloses options for close integration and makes it difficult to merge models later. If integration becomes necessary after the models have diverged, complex translation layers may be required.
- **When to Choose Separate Ways:**
    - **Conflicting Mindsets**: When teams have significantly different views or goals, merging models can be difficult or unproductive.
    - **Specialised Needs**: When different parts of the system have very different requirements and constraints, a unified model may not serve all needs well.
    - **Technical Differences**: When different technologies or tools are used in different parts of the system, integration can be costly and complex.
    - **Limited Integration Needs**: When integration is not needed, or relatively limited, allows continued use of customary terminology and avoids corruption of the models.
    - **Political or Organisational Factors**: Sometimes, political or organisational structures can make it impractical or impossible to unify models.

Open Host Service is described as a strategy for managing integration between different parts of a system, particularly when a subsystem needs to be integrated with many others. This approach involves defining a set of services that other subsystems can access via a published protocol. Instead of creating custom translation layers for each integration, a subsystem exposes a common set of services for others to use.

- **Standardised Protocol**: An Open Host Service defines a protocol that allows other systems to access its functionalities. This protocol acts as a common language, making integration more manageable.
- **Set of Services**: The subsystem is described as a set of SERVICES that cover the common needs of other subsystems. These services provide access to the subsystem's capabilities.
- **Multiple Integrations**: This pattern is particularly useful when a subsystem needs to integrate with numerous other systems. It avoids the maintenance burden of creating and maintaining individual translators for each integration.
- **Flexibility**: The protocol is enhanced and expanded to handle new integration requirements, except for cases where a single team has very specific, idiosyncratic needs. In such cases, a one-off translator can be created to augment the protocol, keeping the shared protocol clean and simple.
- **Shared Model Vocabulary**: The formalisation of communication implies a shared model vocabulary, which is the basis of the service interfaces. This means the other subsystems become coupled to the model of the open host, and other teams are forced to learn the particular dialect used by the host team.
- **Published Language**: In some cases, using a well-known PUBLISHED LANGUAGE as the interchange model can reduce coupling and ease understanding.

- **Trade-offs**: The Open Host Service trades off some flexibility for reduced complexity and maintenance overhead. It aims for a balance between custom solutions and generic integrations.
- **When to Use an Open Host Service:**
    - **High Demand**: When a subsystem needs to integrate with many other systems, especially if the integrations have similar requirements.
    - **Cohesive Services**: When the subsystem's resources can be described as a cohesive set of SERVICES.
    - **Multiple Integrations**: When a significant number of integrations are needed. This approach can be more effective than point-to-point custom solutions.
    - **Need for Consistency**: When there is a need for a consistent way of interacting with the subsystem from various other parts of the overall system.

Published Language is a strategy for managing integration between different parts of a system, or between different systems, by using a well-documented, shared language as a common medium of communication.

Here are the key aspects of a Published Language:

- **Shared Medium**: A Published Language provides a common way to express the necessary domain information for communication between different systems. This can be an existing, well-known standard or one created specifically for the purpose.
- **Translation**: Systems translate their internal models to and from this shared language as needed, allowing them to interact without having to understand each other's specific models. This translation process can be complex, but the goal is to keep the internal models clean and separate from the translation process.
- **Stability**: The Published Language is a stable medium that should not change frequently. Changes to the language can disrupt communication, so it is essential that it is well-documented and robust. It is important to note that the interchange language is not the same thing as the host's domain model and should be kept separate.
- **Documentation:** The language must be well-documented to enable independent interpretations to be compatible. This documentation allows different teams or systems to implement the translation logic correctly and consistently.
- **Reduced Coupling:** By using a common language, systems avoid tight coupling to each other's internal models, making them more flexible and easier to maintain. This contrasts with approaches where systems directly interact with each other's internal representations.
- **Reusability**: Using a well-established language, if available, means that existing tools and expertise can be leveraged, reducing development effort and improving interoperability.
- Examples:
    - Chemical Markup Language (CML): Chemical Markup Language (CML) enables the sharing of complex chemical information, in standard XML based publish language.
    - **DB2 Interface**: well-specified and documented DB2 interface as published language for integrating with another database system, like Btrieve.
- **When to Use a Published Language:**
    - **Multiple Systems**: When several systems need to exchange data and a common language for communication is needed.
    - **Complex Translation**: When direct translation between domain models is complex, a Published Language can simplify the integration by acting as a common intermediary.
    - **Avoiding Model Coupling**: When it's important to prevent tight coupling between systems and allow them to evolve independently.
    - **Industry Standards**: When industry standards or existing published languages are available, they can be reused, reducing the effort of building an interchange language from scratch.
    - **External Integration**: When integrating with external systems, where it may not be possible or desirable to adopt the other system’s domain model directly.

Choosing Right (Your) Model Context Strategy involves making deliberate decisions about how to define and manage the boundaries between different models within a software project. This includes understanding where to apply a unified model and where to allow different models to coexist.

Choosing Right (Your) Model Context Strategy means consciously deciding on the scope of each model and how these models relate. The aim is to create a system where each part is clearly defined and where integration is managed to balance the need for consistency with the benefits of autonomy. This is not a one-time decision, but an ongoing process as the project and its models evolve.

- **Context Map as a Starting Point**: The process begins with an accurate CONTEXT MAP that reflects the current state of the project. This map identifies all the models in use, their boundaries, and their relationships. The map is a snapshot of the project as it is, not necessarily as it should be, and should be updated as the project evolves.
- **Team Involvement**: Teams need to be involved in decisions about where to define BOUNDED CONTEXTS and what type of relationships to have between them. These decisions, or at least an understanding of them, should be propagated throughout the team. These decisions are not just technical but also consider team structures, communication patterns, and business needs.
- **Trade-offs**: Choosing a model context strategy involves balancing several factors:
    - **Value of Independent Action vs. Rich Integration**: The goal is to weigh the benefits of allowing teams to work autonomously with their own models against the advantages of having a highly integrated system.
    - **Seamless Integration vs. Coordination Effort**: A more unified model can offer better integration but requires more effort to coordinate the different teams.
    - **Communication and Control**: Project leaders may need to decide the level of control to exert over the model to allow teams to make decisions independently without fragmenting the model.
    - **Recognizing External Systems**: Some subsystems will clearly not be part of the system under development, such as major legacy systems or external services. These systems should be segregated and treated as separate contexts, often requiring translation layers for integration. However, it's important to be aware that external systems might not conform to the idea of a cohesive BOUNDED CONTEXT with integrated development practices.
    - **Strategies for Relationships**: Once the boundaries are identified, decisions have to be made on the relationships between BOUNDED CONTEXTS:
        1. **Separate Ways**: This is where different parts of a software system use their own models, limiting integration to minimise complexity and allow independent development. This approach is suitable when integration is not essential, or where teams have very different needs and working styles.
        2. **Conformist**: This strategy involves adhering to the model of an upstream team to simplify integration and share a common UBIQUITOUS LANGUAGE. While it limits the downstream team's freedom, it reduces the need for translation layers. This is most useful where a small extension is made to an existing system.
        3. **Anticorruption Layer**: This approach involves creating a translation layer to insulate the system from a poorly designed or incompatible external system. This layer handles the translation between different models and ensures the integrity of the local domain model.
        4. **Shared Kernel**: Parts of the model are shared between contexts, although this can be problematic when different implementation technologies are used.
        5. **Open Host Service**: This strategy involves the creation of a set of services that other subsystems can access via a published protocol rather than each system creating its own custom translation layers. This promotes easier maintenance and consistent interaction.
        6. **Published Language**: This strategy is used to manage integration between different parts of a system, or between different systems, by using a well-documented, shared language as a common medium of communication.
    - **Avoiding Premature Fragmentation**: While it can be tempting to break large models into smaller pieces, it's important to consider whether that is actually necessary and whether the team understands how to maintain and manage many smaller models. There are other approaches to managing large models within a single context .
- **Realistic Transformations**: Transformations of the context map should be pragmatic. Changes to the context map should only be made once the changes in reality are done.

- **Bias Awareness**: When working on a software project, teams tend to focus on the parts of the system they are changing, which might lead to biased CONTEXT MAPS. It's important to be aware of this bias and be mindful of the limits of the applicability of any particular map.

Transformations refers to the deliberate and planned changes to the boundaries and relationships between BOUNDED CONTEXTS within a software project. These transformations are not about minor adjustments but rather significant shifts in how different parts of the system interact and how their models relate.

- **Context Map as a Foundation:** Transformations are based on an existing CONTEXT MAP that accurately reflects the current situation. This map is essential to understand the existing relationships between BOUNDED CONTEXTS before making any changes.
- **Types of Transformations**: The chapter outlines several types of transformations, which often involve breaking up or merging contexts, or changing their relationship patterns. The key transformations discussed are:
    - **Merging Two Contexts into a Shared Kernel**: This involves combining two or more BOUNDED CONTEXTS by identifying a shared subset of their models (a SHARED KERNEL). This can reduce duplication and make integration easier, but it requires careful planning and coordination. The steps to achieve this are:
        1. **Identify the shared subdomain:** Find the parts of the models that are similar or overlapping.
        2. **Create a shared kernel:** Define a new model for the shared domain, including necessary abstractions and interfaces.
        3. **Iterative Integration:** Start with a small SHARED KERNEL and integrate incrementally, addressing translation and dependency issues as they arise.
        4. **Refactor Applications**: Migrate applications to use the new SHARED KERNEL and remove unnecessary translations.
    - **Shifting to a Single Model**: Instead of merging two models, one model is chosen, and the other context is refactored to be compatible with it. This involves transferring full responsibility for subdomains from one context to another and enhancing the chosen model as needed. This simplifies the overall model but might require significant refactoring of existing code. This transition can be long or indefinite, having the pros and cons of going SEPARATE WAYS.
    - **Merging Two Models into a New Model**: This involves creating a new, deeper model capable of handling the responsibilities of both models. This is the most ambitious approach and often the best for the long-term, but requires a strong understanding of the domain.
    - **Replacing Legacy Systems:** When a legacy system is being replaced, the functionality should be migrated in small increments into the new system. Steps include:
        1. Identify functionality to add to the new system in each iteration.
        2. Identify necessary additions to the ANTICORRUPTION LAYER.
        3. Implement and deploy in each step.
    - **Transforming to a Published Language:** A PUBLISHED LANGUAGE can be created when there is a need to integrate with multiple systems. Steps include:
        1. Select a core domain model to serve as the base language.
        2. Use a standard format, such as XML, to create the language.
        3. Publish the language and system architecture.
        4. Build translation layers for all collaborating systems.
        5. Switch over to the new language.
- **Incremental Approach:** Transformations are rarely completed in a single step. They are typically achieved through multiple iterations, where changes are implemented, tested, and refined. Each step should be manageable and the team must be careful not to take on too much at once.
- **Team Involvement:** Transforming BOUNDED CONTEXTS may require team members to move between teams to share their knowledge.
- **Continuous Integration:** As part of any transformation, the process of CONTINUOUS INTEGRATION must be maintained in order to keep the team working together. This also includes making sure all the components fit within the large-scale structure.

- **Distillation:** Distillation techniques can be used to refine each model prior to merging them.
- **Pragmatic Choices:** Transformations may be influenced by business and organisational factors and political issues. The teams may not get what they want and have to assess the costs of a transformation. It's important to be pragmatic about which transformations to undertake and how far to push them.
- **Careful Implementation:** It is critical not to change the map until a real change to the models is done. Teams should work through any contradictions and fix real problems before changing the CONTEXT MAP.
- **Awareness of Costs:** Teams should be aware that there are costs involved in any transformation. Merging two contexts can reduce duplication, but may also impose a burden of coordination between teams. Going SEPARATE WAYS can allow independent development but may require more complex translation layers later. Transformations should be based on the value they bring to the system.

Chapter 15: Distillation

Distillation is the process of separating the components of a mixture to extract the essence in a form that makes it more valuable and useful.

Core Domain

- **Core Domain is the most important part of a software system's** model. It is the part that is distinctive and central to the users' goals. The Core Domain is where the most value should be added to the system and where the most specialized and differentiating aspects of the application reside.
- **Central to User Goals**: The Core Domain is directly related to the primary objectives of the users and the business problems the software is intended to solve.
- **Differentiating Factor**: It's what makes the application unique and gives it a competitive advantage. It is where the most specialized knowledge and business rules are implemented.
- **Value Driver**: The Core Domain is the area where the most effort and talent should be focused, with other parts of the system supporting it.
- **Distillation**: Identifying the Core Domain involves a process of distillation, where the essential parts of the model are extracted from the more generic and less important aspects.
- **Not Always Obvious**: What constitutes the Core Domain depends on the application's specific needs and may not be apparent at first, requiring iterations of knowledge crunching and refactoring.
- **Evolving Concept**: Like any other part of the design, the identification of the Core Domain should evolve through iterations.
- **Focus of Resources**: The Core Domain should receive the most attention and resources, with the most skilled developers working on it.
- **How to build the team:** a set of strong developers who have a long-term commitment and an interest in becoming repositories of domain knowledge with one or more domain experts who know the business deeply.

Escalation of Distillations refers to a series of techniques that can be applied, with increasing commitment and impact, to refine a domain model and focus on its core elements. These techniques range from simple documentation to significant restructuring of the model and code.

- **Domain Vision Statement**: This is the initial step, involving the creation of a short document (around one page) that describes the Core Domain and its value. It highlights the essential aspects of the model and how they serve diverse interests, without going into technical detail or modifying the model directly. This statement acts as a guide for the development team.
- **Highlighted Core**: This involves identifying and flagging the elements of the Core Domain within the primary repository of the model. It makes the core easily visible to developers without requiring extensive changes to the model or code. The goal is to make it effortless for a developer to know what is in or out of the Core Domain. This can be achieved using techniques such as highlighting, stereotypes, comments, or development environment tools.
- **Generic Subdomains**: At this stage, the process becomes more active, involving the separation of Generic Subdomains from the Core Domain. Generic Subdomains are parts of the model that add complexity without capturing specialized knowledge. These are factored out into separate modules, leaving the Core Domain

more focused and easier to understand. This separation helps to clarify the meaning of both the core and generic elements.

- **Cohesive Mechanisms**: This involves encapsulating complex algorithms or processes into separate frameworks with intention-revealing interfaces. This reduces the complexity of the core domain by shifting the mechanistic "how" into a dedicated framework, allowing the core to focus on the "what.".
- **Segregated Core**: This step involves restructuring the model and code to make the Core Domain directly visible. Related classes are moved into a new module named for their core concept, and code is refactored to remove non-core responsibilities. This creates a clear separation between the Core Domain and other parts of the system. The refactoring also makes references to other packages explicit.
- **Abstract Core**: This is the most ambitious step, where the most fundamental concepts and relationships are factored into distinct classes, abstract classes, or interfaces. This creates a pure form of the model that expresses most of the interaction between significant components. The Abstract Core is placed in its own module, with specialized implementation classes left in their respective subdomains.

Each level of distillation involves a greater commitment and leads to a sharper focus on the Core Domain, resulting in a more manageable, expressive, and valuable model.

Generic Subdomains are parts of a software system's model that add complexity without capturing or communicating specialised knowledge. They are essential to the functioning of the system, but they are not the core focus of the project.

- **Not a primary focus**: Generic subdomains represent concepts and functionalities that are needed by many businesses and applications and are not unique to a particular domain. They support the core domain but are not the driving force behind the software.
- **Common Knowledge**: They involve general principles or details that are widely known or belong to specialties that are not the project's main focus.
- **Essential but not Differentiating**: While necessary for the system to function, generic subdomains do not provide a competitive advantage or distinguish the application from others.
- **Examples**: Examples of generic subdomains include models for corporate organisation charts, accounting, and date/time management.
- **Separation from the Core**: Generic subdomains should be separated from the core domain to make it easier to understand and manage the most valuable parts of the system.
- **Lower Development Priority**: Once separated, generic subdomains should be given lower development priority compared to the core domain. Core developers should not be assigned to these tasks, as they will not gain much domain knowledge.
- **Potential for Off-the-Shelf Solutions**: Off-the-shelf solutions or published models may be considered for generic subdomains to reduce development effort.
- **Not Necessarily Reusable**: While generic subdomains might seem like good candidates for code reuse, the primary focus should be on keeping them generic, not on making them reusable.
- **Model Reuse**: Model reuse is often better than code reuse.
- **Why are Generic Subdomains Important?**
    - **Clarity**: By separating out generic subdomains, you make the core domain more prominent and easier to understand.
    - **Focus**: It allows the development team to focus on the unique aspects of the application, which provide the most value.
    - **Resource Management**: It enables a project to allocate resources more efficiently, with top talent focused on the core domain.

The following are the different ways to develop Generic Subdomains

- **Off-the-Shelf Solutions:** Purchasing pre-built or open-source implementations for the generic subdomain
    - **Advantage**: Reduces code development. Externalises maintenance burden. Code may be more mature and complete.

    - **Disadvantages**: Requires evaluation and understanding before use. Quality may be inconsistent. May be over-engineered. Integration issues may arise. Potential for platform or compiler dependencies.
- **Published Design or Model:** Using widely distributed, documented models such as those in analysis patterns or formalised models for the generic subdomain.
    - **Advantages:** Uses mature models refined by many insights. Instant, high-quality documentation available.
    - **Disadvantages:** Model might not perfectly fit project needs. Model may be over-engineered.
- **An Outsourced Implementation:** Outsourced the development of the generic subdomains.
    - **Advantages:** Keeps CORE team free to work on CORE model. Faster development, without need to maintain a large team for long. An interface-oriented design approach can be leveraged ,
    - **Disadvantages**: Need CORE team time to overlook the development and design prospects. Need transitioning back to CORE on completion. Varying code quality.
- **In-House Implementation:** Developing the generic subdomain in-house, either by the core team or by a separate team.
    - **Advantages:** Allows for easy integration. Provides exactly what is needed, without extra features. Temporary contractors can be assigned.
    - **Disadvantages:** Creates ongoing maintenance and training burden. Time and cost of development can be underestimated.
- **Combination of Approaches:** Combining some of the above approaches; for example, using a published model as a basis for an in-house implementation.
    - **Advantages:** Flexibility in using the best aspects of different approaches.
    - **Disadvantages:** Must carefully consider the trade-offs of each approach to ensure they combine effectively.

Domain Vision Statement : A Domain Vision Statement is a short, concise description of the core domain and its value to the business. It's a strategic tool used in domain-driven design to focus development efforts and guide decision-making. The statement is designed to be easily understood by both technical and non-technical team members, management, and even customers.

- **Focus on the Core Domain**: It highlights the unique and valuable aspects of the domain model. It intentionally omits details that don't differentiate the application from others.
- **Value Proposition**: The statement explains how the domain model will provide value to the organisation. It articulates the specific benefits the application will bring.
- **Balances Diverse Interests**: If the domain model serves multiple stakeholders, the statement shows how their interests are addressed and balanced.
- **Concise**: A Domain Vision Statement is typically about one page in length. This brevity ensures that it can be easily read and understood by all involved parties.
- **Living Document**: It's written early in the project and revised as new insights are gained. This iterative approach ensures the statement remains relevant and aligned with the evolving understanding of the domain.

The purpose of a Domain Vision Statement is to:

- **Guide Development:** It acts as a guidepost, ensuring the development team is headed in a common direction. This common direction helps to keep the project focused on the most critical elements of the domain.
- **Aid Resource Allocation**: It helps to direct resources to the most crucial points in the model and design. By making it clear which aspects of the system are most important, it enables more effective resource management.
- **Improve Communication**: It facilitates communication among team members, management, and customers. A shared vision makes it easier to discuss the project and understand priorities.
- **Educate Team Members:** It serves as an educational tool for new team members, quickly bringing them up to speed on the critical aspects of the domain. It provides a shared understanding and knowledge base.

- **The Domain Vision Statement is not meant to be a detailed technical specificatio**n or a complete design document. Instead, it is intended as a high-level, easily digestible summary of the project's core purpose and value within its specific domain.

The "**Highlighted Core**" concept, is a technique used to make the **most important parts of a domain model more visible**. The "Highlighted Core" aims to make the core domain easily recognizable, and can be implemented in a few different ways:

- **Distillation Document:** This involves creating a brief document (3 to 7 pages) describing the core domain and its primary interactions. It acts as a minimalist entry point that explains the core and suggests areas for closer scrutiny, guiding the reader to the appropriate parts of the code. The document is not a complete design document, but a starting point.
- **Flagged Core**: This involves marking the elements of the core domain directly within the primary repository of the model, without trying to explain its role. This can be achieved in various ways, such as using a "stereotype" in UML diagrams, comments in code, or a tool in the development environment. The key is that a developer can easily see what is part of the core domain, and what is not.

The "**Highlighted Core**" is a reflection on the model itself and not necessarily part of the model. It is intended to be a low effort way to make the core domain more visible. The "Highlighted Core" supplements other more aggressive techniques such as partitioning generic subdomains and allows teams to more effectively refactor and manage the core domain.

The **distillation document** can also serve as a guide for when to consult with other team members. When developers realise that the distillation document itself requires a change to stay in sync with their code or model change, consultation is needed. Changes to the core domain will have a large effect, and this can be easily identified when the distillation document needs updating. This allows developers to maintain autonomy when changes are made outside of the core, or when changes are made to details not included in the distillation document.

Cohesive Mechanisms refers to the practice of partitioning conceptually cohesive computational processes into separate, lightweight frameworks. This is done to prevent complex algorithms from bloating the design and obscuring the "what" with the "how".

*Implementation***:** The framework should expose its capabilities through an intention-revealing interface, which allows other domain elements to use the mechanism in a more declarative style. This approach enables the core domain to focus on expressing the problem ("what") and delegate the intricacies of the solution ("how") to the framework.

Key Characteristics and Benefits of Cohesive Mechanisms:

- **Focus on Computation**: They are narrowly focused on solving a computational problem, not on representing the domain.
- **Separation of Concerns**: They separate the "what" (the problem) from the "how" (the solution), leading to a cleaner design.
- **Reusability**: They can be based on well-documented algorithms or formalisms, making them easier to implement and maintain.
- **Clarity**: They allow the core domain model to be more expressive and less cluttered by complex implementation details.
- **Declarative Style**: They allow the core domain to make meaningful statements by utilizing well defined interfaces, rather than calling obscure functions.
- **Reduced Coupling**: The mechanism becomes encapsulated, which decouples the core domain from specific methods of solving problems.

Examples of Cohesive Mechanisms:

- **Graph Traversal Framework**: As an example of this, a graph traversal framework could be used to navigate an organisational structure, by focusing specifically on the traversal algorithms. The organisation model can then simply state how people and relationships should be interpreted as nodes and edges, and delegate the traversal of the structure to the graph framework.

- **Specification Framework**: A framework for creating SPECIFICATION objects and supporting comparison and combination operations can also be a cohesive mechanism. The core domain can then declare its SPECIFICATIONS using the framework, and leave the complexities of comparisons and combinations to the framework.
- **Shares Math**: An example given in the source material is of a "Shares Math" framework that was created to handle the complexities of how to distribute loan payments to lenders. This framework allowed for the core domain to focus on the overall loan process, and delegate the complexities of the pro-rata distribution calculations to the "Shares Math" framework.
- **Custom Algorithms**: If a specific algorithm is proprietary and a key part of the value of the software, then it may be considered part of the core domain.

Relationship to Generic Subdomains:

- Both Generic Subdomains and Cohesive Mechanisms aim to reduce the burden on the core domain.
- Generic Subdomains are based on expressive models that represent aspects of the domain, but are not central to the project.
- Cohesive Mechanisms focus on solving computational problems and do not typically represent domain concepts.

In practice, this distinction may not always be clear, and through refactoring a mechanism may be distilled into a purer form or transformed into a generic subdomain.Cohesive Mechanisms are a way to manage complexity by separating out well-defined, reusable computations from the core domain model. This promotes a cleaner, more understandable, and maintainable design.

Segregated Core involves refactoring a domain model to explicitly separate core concepts from supporting elements, enhancing the cohesion of the core and reducing its coupling to other parts of the system. This technique addresses the issue of the core domain being obscured and entangled with less critical aspects of the model, which hinders understanding and design.

- **Problem:** In complex systems, core domain concepts can become entangled with supporting elements, making it difficult to understand the most important relationships and leading to a weak design. Even after factoring out generic subdomains, the core might remain cluttered and lack strong conceptual cohesion.
- **Solution:** Refactor the model to isolate the core concepts into their own modules. This involves moving related classes into a new module, and then removing data and functionality that aren't directly related to those core concepts. These removed aspects are placed into other, related packages. The focus is on strengthening the cohesion of the core, and simplifying its interactions with other modules.
- **Goal**: To create a clear and manageable core domain that is easy to work with and understand, and to give the core domain more focus, making it more expressive.

Key Characteristics and Benefits of a Segregated Core:

- **Enhanced Cohesion**: The core domain's conceptual integrity is strengthened by grouping related elements together and separating them from the rest of the system.
- **Reduced Coupling**: By separating the core from supporting elements, the dependencies between them are reduced, making the core easier to change and evolve.
- **Improved Visibility**: The most critical parts of the model are made more visible, which helps designers understand the system's essence and prioritise their efforts.
- **Focused Development**: By making the core domain more explicit, development efforts can be focused on the most valuable aspects of the system.
- **Clear Relationships**: The relationships between the core domain and other parts of the system are made explicit and self-explanatory, further aiding understanding.

Steps to Create a Segregated Core:

1. **Identify the Core Subdomain:** Begin by identifying the most crucial parts of the model, often drawing from the Domain Vision Statement.

2. **Move Related Classes**: Transfer all classes related to the core subdomain into a new module, named to reflect the core concept.
3. **Refactor Code**: Remove any data and functionality that aren't directly related to the core concept. These removed aspects are then placed in other packages with conceptually related tasks.
4. **Refactor the Segregated Core Module**: Simplify the relationships and interactions of the new module, and minimise and clarify its connections with other modules.
5. **Repeat**: Apply this process to other core subdomains until the entire core has been segregated.

Costs of Creating a Segregated Core:

1. **Increased Complexity of Non-Core Relationships**: Separating the core may make some relationships with tightly coupled non-core classes more complicated or obscure.
2. **Significant Effort**: Segregating the core requires substantial refactoring effort throughout the system.

When to Apply a Segregated Core:

- When you have a large, critical bounded context that is difficult to understand and manage as a whole.
- When the core domain's essential parts are obscured by a lot of supporting code and functionality.

Example:

The sources use the example of a cargo shipping model, where the core focus is on the reliable delivery of cargo according to customer requirements. Initially, the model includes concepts like Customer, Cargo, Customer Agreement, Handling Step and other supporting features. The Delivery module is created as a segregated core, containing classes like Cargo, Customer Agreement and Handling Step, whilst the other concepts are relegated to a supporting role. The Customer class is then removed from the core as it is not essential to the delivery of cargo at the operations level. This segregation of the core allows the team to focus on the primary goal of reliable delivery and customer requirements.

Relationship to Other Concepts:

- **Distillation**: Segregated core is a part of the distillation process, which aims to extract the essence of the system and make it more valuable. This is achieved by separating concerns.
- **Generic Subdomains**: Segregation of the core often happens after Generic Subdomains have been identified and separated from the system.

Abstract Core is a technique used to distill the most fundamental concepts of a domain model into a separate module, specifically when a model becomes too complex to grasp, even after applying other distillation techniques. It's a way of managing complexity and enhancing understanding by focusing on the core abstractions that define the interactions between significant components of the system.

Identify the most fundamental concepts in the model and factor them into distinct classes, abstract classes or interfaces. These abstractions should then be placed in their own module, separate from the more specialized implementation classes. This approach shifts the focus from concrete implementations to the core abstractions that drive the system. The goal is to create an abstract model that expresses most of the interaction between significant components, providing a clear and succinct view of the main concepts and their relationships.

The abstract core should resemble the distillation document (if one was used), which shows the fundamental concepts and relationships of the system.

Deep Models: The abstract core enables a deep model as it distills the most essential aspects of a domain into simple elements that can be combined to solve the important problems of the application. This process is not limited to just separating parts of the domain away from the core but also includes the continuous refinement of those domains.

Choosing Refactoring Targets addresses the challenge of where to begin when faced with a large, poorly factored system. Instead of aimless refactoring, this concept provides a strategic approach to decide which parts of the system to improve, to achieve the most impact.

Refactoring Approaches:

- Pain-Driven Refactoring: When refactoring in response to a specific problem (a "pain point"), investigate whether the root cause involves the core domain or its relationship to a supporting element. If so, address the root cause within the core domain first.
- Proactive Refactoring: When refactoring freely (i.e. not in response to a specific problem), focus first on better factoring of the core domain, improving its segregation from other parts of the system, and purifying supporting subdomains to make them more generic. This could be in preparation for new functionality or to ensure code quality.

Chapter 16: Large-Scale Structure

Large-Scale Structure discusses the need for a high-level organising principle to understand a system as a whole, beyond modularity. It introduces patterns of rules, roles, and relationships that span an entire system, guiding design and assisting in the coordination of independent work.

The below are the different approaches to structuring a design at this level, emphasising the importance of an evolving structure based on a deep understanding of the domain:

Evolving Order is a principle that advocates for the evolution of a system's conceptual large-scale structure alongside the application itself. Rather than imposing a rigid, pre-defined structure, this approach recognises that a suitable structure emerges through the development process.

Key aspects of Evolving Order include:

- **Avoiding up-front design constraints**: It discourages the freezing of design decisions early in the project, as requirements change and understanding deepens.
- **Flexibility and adaptation**: The structure is not static and should be allowed to change over time, possibly even to a completely different type of structure. This allows for a better fit to the domain and requirements.
- **Developer empowerment**: It aims to avoid architectures that take too much power away from application developers and enables them to create designs and models that work well for the specifics of the problem.
- **Continuous refinement**: The large-scale structure should be continuously re-evaluated and modified as understanding of the domain grows. This includes the possibility of completely changing the structure as the project evolves.
- **Organic growth**: It promotes the idea that order can emerge organically from a team's collective insight, not necessarily through central authority, allowing a shared set of principles to develop over time.
- **Learning through application**: The practical application of the structure will reveal what works and what doesn't, allowing it to be refined based on real experience.

By allowing the large-scale structure to evolve, the development team can avoid the trap of being constrained by an initial design that doesn't fully align with the domain or the changing requirements. This approach is particularly useful when the understanding of the problem is not clear at the outset of the project. The concept of Evolving Order is applicable across multiple BOUNDED CONTEXTS.

System Metaphor is a way of using a concrete analogy to bring order to the development of a whole system. It provides a shared understanding among all team members by using a central theme of a design that is easy to understand.

Key aspects of a System Metaphor include:

- **Tangible Understanding**: Software designs can be very abstract and difficult to grasp, so developers and users need tangible ways to understand the system and share a view of the system as a whole.
- **Shared Understanding**: A System Metaphor conveys the central theme of a whole design, and provides a shared understanding among all team members.
- **Guiding Development**: Developers make design decisions that are consistent with the system metaphor. This consistency enables other developers to interpret different parts of a complex system in terms of the same metaphor.
- **Reference Point**: The metaphor can act as a reference point in discussions for both developers and experts, and may be more concrete than the model itself.

- **Loose Structure**: A System Metaphor is a loose, easily understood, large-scale structure that is harmonious with the object paradigm.
- **Application in Multiple Contexts**: Because a System Metaphor is only an analogy to the domain, different models can map to it in an approximate way. This allows it to be applied in multiple BOUNDED CONTEXTS, helping to coordinate work between them.
- **Potential Risks**: A persuasive metaphor introduces the risk that the design will take on aspects of the analogy that are not desirable for the problem at hand, or that the analogy may not be appropriate. It is important to continually re-examine the metaphor for over-extension or inaptness, and be ready to drop it if it gets in the way.

Example: "firewall" as a System Metaphor. This metaphor has influenced network architectures and shaped a whole product category. The shared understanding of a firewall is due, in no small part, to the metaphor.

Responsibility Layers is presented as a way to structure a large system by grouping elements based on their roles and dependencies. Instead of just separating concerns, Responsibility Layers aim to identify natural strata within a domain, based on how concepts relate and how they change, and then organise the design accordingly.

Key aspects of Responsibility Layers include:

- **Conceptual Dependencies**: Layers are defined by the relationships between concepts. For example, concepts in a higher layer depend on those in lower layers, while lower-layer concepts should be meaningful on their own. This is also described as an awareness of and ability to use services of layers "below", but not above.
- **Natural Stratification**: The layers reflect the natural divisions within a domain, based on how different parts change and for what reasons. This is related to the idea of Conceptual Contours, where layers align with different rates or sources of change.
- **Broad Responsibilities**: Unlike individual objects, each layer is assigned a broad, abstract responsibility that encompasses multiple objects, aggregates and modules. These responsibilities should tell a story of the system's purpose.
- **Layering Principle**: The pattern of layering used is a Relaxed Layered System, where elements of a layer can access any lower layer, not just the one immediately below.
- **Guiding Design**: As individual modules and aggregates are designed, they are factored to keep them within the bounds of a major responsibility associated with one of the layers.
- **Evolutionary**: The layers are not fixed, and as the understanding of the domain evolves so does the structure of the layers. This supports the idea of Evolving Order.

Figure 16.13. Conceptual dependencies and shearing points in a factory automation system

- Initially, two layers are identified: **Operations** (what is being done, such as tracking cargo) and **Capability** (the resources used, like transport schedules).
- A third layer, **Decision** Support (tools for planning and decision making, like a route planner), is later added.

The example shows how the team had to refactor their model to move a "preferred" attribute from the transport leg to a separate policy, in order to fit the new structure of the three layers. When a new feature is added for hazardous material routing, the team needed to choose an implementation that was consistent with the established layers, and avoided having a dependency from the Operations layer to the Decision Support layer. Responsibility Layers dictate a kind of factoring of model concepts and their dependencies, and this allows rules that can specify communication patterns between the layers.

- Potential: Defines the resources and capabilities of the organisation.
- Operation: Defines the actions, activities and the realities of the situation.
- Decision Support: Provides tools for planning and decision making.
- Policy: Defines the rules and goals that constrain behaviour in other layers.
- Commitment: Defines the goals that direct future operations.

Responsibility Layers provide a structure for large systems by grouping related concepts and responsibilities into distinct layers, reflecting the natural structure of the domain and guiding the design decisions that must be made throughout the development process. This approach aims to improve the comprehensibility of the system and facilitate coordination between team members working on different parts of the project. The choice of appropriate layers is a business modelling decision, rather than a technical one.

Knowledge Level introduces as a way to manage complexity in systems where the rules, roles, and relationships between entities can change. It is a structure that separates the description of how a system should behave from the operational behaviour itself.

Key aspects of the Knowledge Level include:

- **Separation of Concerns**: The Knowledge Level separates the "*self-defining*" aspects of the model, making constraints explicit. This separates the rules and policies from the objects they govern.
- **Configurable Behaviour**: It addresses the requirements for software with configurable behaviour, where roles and relationships among entities must be changed at installation or even at runtime.
- **Flexibility with Constraints**: Instead of either a rigid model or a fully flexible one that allows any relationship, the Knowledge Level allows users to configure the software to reflect the current structure of the organisation while also enforcing its rules.
- **Distinct Set of Objects**: A distinct set of objects are created to describe and constrain the structure and behavior of the basic model. The concerns of these two levels are kept separate, one very concrete and the other reflecting rules and knowledge that a user or superuser is able to customise.
- **Specialised Constraints**: It does not strive for full generality, as a very specialised set of constraints on a set of objects and their relationships can be more useful than a generalised framework. This specialisation helps to communicate the specific intent of the designer.
- **Reflection Pattern**: The Knowledge Level applies the Reflection pattern to the domain layer. Reflection makes software "*self-aware*" by making aspects of its structure and behaviour accessible for adaptation.
- **Mutual Dependencies**: Unlike Responsibility Layers, dependencies run in both directions between the levels.
- **Not a Layer**: Although it may resemble layering, Reflection involves mutual dependencies running in both directions.

Knowledge Level example - employee payroll and pension system:

- Initially, the model combines concepts of an employee's type and their payroll details within the same object.
- The team recognises that certain objects, such as the Employee Type, have restricted access while others, such as the Employee, can be freely edited.
- The team then refactors the model so that the Employee Type imposes behaviour on the Employee. The restricted edits are in the Knowledge Level, while day-to-day edits are in the operational level.

This insight reveals that two distinct concepts (Employee Type and Payroll) are being combined within the same object, which leads to the factoring out of Payroll into its own object. Knowledge Levels can be used in conjunction with other large-scale structures such as Responsibility Layers, and provides a structure that is not part of those other structures. The Knowledge Level provides a way to create systems that are both flexible and maintainable by separating the rules and constraints from the operational logic of the application, allowing these to be customised by users while still enforcing necessary rules. It also enables a deeper understanding of the underlying structure and relationships within the system, leading to further insights and refinements.

Pluggable Component Framework is a way to structure a system that allows for diverse implementations of interfaces to be freely substituted. This approach is most useful in mature models where multiple applications need to interoperate based on the same abstractions, but have been developed independently.

Key aspects of a Pluggable Component Framework include:

- **Abstract Core**: A central abstract core of interfaces and interactions is defined. This core is a shared understanding of the key concepts.
- **Diverse Implementations**: The framework allows for diverse implementations of those interfaces to be substituted freely. This is the pluggable nature of the framework.
- **Central Hub**: Components plug into a central hub, which supports any protocols they need and knows how to interact with the interfaces they provide. Other connection patterns are also possible.
- **Component Responsibility**: Each component has responsibility for certain categories of functions.
- **Independence**: While the design of the interfaces and the hub must be coordinated, the interiors of the components can be designed with greater independence. This allows for different development teams to work independently on specific components.
- **Shared Kernel**: The central hub is an Abstract Core within a Shared Kernel.
- **Multiple Contexts**: Multiple Bounded Contexts can exist behind the encapsulated component interfaces, which makes the framework especially useful when components are coming from different sources, or when integrating pre-existing software.
- **Flexibility**: Any application can use the components as long as they operate through the interfaces of the Abstract Core.
- **Published Language**: In some cases, a Published Language can be used for the plug-in interface of the hub.

SEMATECH CIM Framework Example :

- The CIM framework defines abstract interfaces for the basic concepts of semiconductor manufacturing execution systems (MES), such as 'Process Machine'.
- Vendors can develop specialised implementations of these interfaces, such as a machine-control component that adheres to the Process Machine interface, and these components can then plug into any application based on the CIM Framework.
- The framework defines rules for how these interfaces can interact within an application, ensuring that any compliant application can count on the services of those interfaces.
- The framework is coupled with CORBA to provide persistence, transactions, and events.

Challenges with Pluggable Component Framework:

- **Difficulty in Application**: It is a difficult pattern to apply, requiring precise interface design and a deep model to capture the required behaviour in the Abstract Core.
- **Limited Options**: Applications have limited options, and if they need a very different approach to the Core Domain then the structure will be restrictive.
- **Frozen Refinement**: Continuous refinement of the core and refactoring is limited as the Abstract Core is frozen due to the need to maintain the protocol of diverse components.
- **Maturity**: The successful application of this pattern relies on a deep understanding of the domain and is more appropriate for later stages of development, not at the start of a project.

How Restrictive Should a Structure Be?

Exploring the trade-offs between imposing strict rules and allowing flexibility when designing a large system. A large-scale structure should provide conceptual coherence and insight into the domain, but must not become overly prescriptive and hinder development.

- **Range of Restrictiveness**: The large-scale structure patterns discussed in the chapter range from very loose, such as the System Metaphor, to very restrictive, such as the Pluggable Component Framework.
- **Trade-off Between Uniformity and Flexibility**: A more restrictive structure increases uniformity and makes the design easier to interpret, as it pushes developers towards good designs and ensures disparate pieces fit together better. However, it may also take away the flexibility developers need and make it difficult to apply across different Bounded Contexts, especially in heterogeneous systems.
- **Conceptual Coherence**: The most important contribution of a large-scale structure is conceptual coherence and giving insight into the domain. Therefore, the structure's rules should make development easier, not more difficult.
- **Avoiding Over-Engineering**: It is important to fight the temptation to build frameworks and regiment the implementation of the large-scale structure.

- **Communication Patterns**: More restrictive structures may specify communication patterns between layers. For example, in a manufacturing plant, an Operations layer might use an event mechanism to communicate changes to a higher Policy layer, without creating two-way dependencies.
- **Framework Temptation**: There is a temptation to build frameworks to enforce a structure, but the primary goal of a large-scale structure is conceptual coherence and understanding of the domain.
- **Minimalism**: It is important to keep the structure simple and lightweight, addressing only the most serious concerns and leaving the rest to be handled on a case-by-case basis.
- **Evolving Order**: A team must be committed to rethinking the large-scale structure as the project evolves, so as to avoid being restricted by an ill-fitting or out-of-date structure.
- **Application of the structure**: The structure must be understood by the entire team, the terminology must enter the Ubiquitous Language, and the team must use self-discipline in its application.

Finding a right balance between the need for consistency and the need for flexibility. Ultimately, the appropriate level of restrictiveness depends on the specific project and its requirements. A large-scale structure should be evaluated throughout the project to ensure that it still fits the problem and is not creating more problems than it is solving. The best approach is to start with a minimal, loose structure and gradually evolve it based on a deeper understanding of the domain and the needs of the project.

Refactoring Toward a Fitting Structure address how to manage the evolution of a large-scale structure within a software project. It acknowledges that the ideal structure is unlikely to be known at the outset and must emerge through an iterative process of learning and adaptation. This concept emphasises that a team must be willing to rethink the structure throughout the project lifecycle and must not be restricted by an ill-fitting structure conceived early in the project.

- **Evolving Order**: A large-scale structure should evolve with the application, and it may change to a different type of structure along the way. The structure should not over-constrain the detailed design and model decisions that must be made with detailed knowledge.
- **Iterative Process**: The development process must be iterative in order to arrive at a fitting structure. This means that the structure should be refactored as the understanding of the domain deepens.
- **Deep Understanding**: A useful structure is only found through a very deep understanding of the domain and the problem. This deep understanding of the domain is achieved through an iterative development process.
- **Fearless Rethinking**: A team must fearlessly rethink the large-scale structure throughout the project life cycle.
- **Cost of Change**: Changes to the structure may lead to a lot of refactoring, as the entire system needs to adhere to the new order.
- **Benefits of Structure**: A design with a large-scale structure is usually much easier to transform than one without. This seems to be true even when changing from one kind of structure to another.
- **Minimalism**: To keep costs down, the structure should be kept simple and lightweight. It should not attempt to be comprehensive, but should address the most serious concerns, leaving the rest to be handled on a case-by-case basis.
- **Communication**: The entire team must understand the structure for it to be followed during development and refactoring, and its terminology and relationships must enter the Ubiquitous Language. The team must use self-discipline in applying the structure.
- **Distillation**: Continuous distillation should be applied to the model to reduce the difficulty of changing the structure. By removing mechanisms, Generic Subdomains, and other supporting structure from the Core Domain, there will be less to restructure.
- **Supporting Elements**: Supporting elements should be defined to fit into the large-scale structure in a simple way. They may have to be refactored to find their place in the structure.
- **Clarity**: The structure should allow people to see deep into the design, which makes manipulation of the system on a large scale easier and safer.

Chapter 17: Bringing the strategy together

Domain-Driven Design synthesizes the strategic design principles and techniques discussed earlier. It addresses how to combine these concepts in large, complex systems, providing guidance on devising a design strategy.

- **Combining Large-Scale Structures and Bounded Contexts**: A local structure can be useful in a very complicated but unified model, raising the complexity ceiling on how much can be maintained in a single BOUNDED CONTEXT.
- **Application of Terminology**: The terminology of the structure applies to the whole project or at least some clearly bounded part of it.
- **Leveraging Multiple Structures**: It is possible to use different structures to organise models within different BOUNDED CONTEXTS, but this may reduce the unifying power of a large-scale structure.
- **Complementary Nature**: Large-scale structure and distillation are complementary. The large-scale structure can help explain the relationships within the CORE DOMAIN and between GENERIC SUBDOMAINS.
- **Distilling Insights**: Distinguishing layers such as potential, operations, policy, and decision support can distill insights fundamental to the business problem, especially in projects with many BOUNDED CONTEXTS.

Six Essentials for Strategic Design Decision Making

1. **Decisions must reach the entire team**: For a strategy to be relevant, everyone needs to be aware of it and adhere to it. A strategic design that emerges from the application team may be more effectively communicated and adopted, provided there is very good communication within the team.
2. **The decision process must absorb feedback**: A deep understanding of the project's needs and the domain's concepts is necessary for creating an organizing principle, large-scale structure, or distillation. Application development team members possess this depth of knowledge.
3. **The plan must allow for evolution**: Strategic design should not be treated as a fixed deliverable but rather as something that evolves during the project. This evolution requires adaptability in project organisation, team structure, and technical choices.
4. **Architecture teams must not siphon off all the best and brightest**: It is essential to have strong designers on all application teams and to include domain knowledge on any team attempting strategic design.
5. **Strategic design requires minimalism and humility**: Strategic design requires organising principles and core models that are pared down to contain nothing that does not significantly improve the clarity of the design.
6. **Objects are specialists; developers are generalists**: Good object design involves giving each object a clear and narrow responsibility and reducing interdependence to an absolute minimum. A good project should encourage developers to be generalists, with everyone communicating and collaborating.

New words / Idioms

- *Corollaries* - a proposition that follows from (and is often appended to) one already proved.
- Profoundly – Greatly.
- Pervasively – spreading through every parts of something, OR existing within.
- Addendum – As added thing, supplement to a book.
- Eerie – Strange and frightening
- **Intrinsic – Belonging natural**
- Unencumber – Not having any burden, impediment.
- Tractable - easy to control or influence.
- Conform: comply with rules, standards, or laws.
- Straitjacket - a strong garment with long sleeves which can be tied together to confine the arms of a violent prisoner or mental patient.
- Immutable – unchangeable.
- Contours – An outline representing or bounding the shape or form of something.
- A "***rat's nest of dependencies***" refers to a complex and tangled web of interconnected dependencies between different software components, where one element relies on many others, creating a difficult-to-manage and potentially unstable system, much like how a real rat's nest appears messy and intertwined.
- "***Don't throw the baby out with the bathwater***" is an idiom that means to be careful not to lose something valuable while getting rid of something unwanted. It's important to separate the good from the bad.
- **Subsumption:** The act of placing something under a more general category. For example, "Soldiers from many different countries have been subsumed into the United Nations peace-keeping force".
- *Rigour : The quality of being extremely thorough and careful. For Example : "his analysis is lacking in rigour".*
- Supple : *bending and moving easily and gracefully; flexible.*
- *Intertwined. - twist or twine together.*

- *Nitpick, find or point out minor faults in a fussy or pedantic way.*
- *"**Mull it over**" is a phrasal verb that means to think about something carefully and for a long time, often before making a decision.*
- *"**Take another stab at it**" means to attempt something again or to try it once more.*
- *The idiom "**wag the dog**" means to divert attention from a more important issue to a less important one. It's often used to describe when a government or other powerful entity tries to manipulate public opinion.*
- *"**knife gets sharper as its blade is ground finer**"*

New Book

- *Analysis Patterns [Tech. Term] [reference: Fowler's Analysis Patterns book]*