Strategic Design

Parinya Ekparinya

Parinya.Ek@kmitl.ac.th

The Development of Large Software Systems

- While this lecture is mainly about large projects which require the combined efforts of multiple teams, the same concepts can be applied to small/medium size projects that involve many subdomains and/or require many models in different contexts.
- We are faced with a different set of challenges when multiple teams, under different management and coordination, are set on the task of developing a project.
- Enterprise projects are usually large projects, which employ various technologies and resources.
- The design of such projects should still be based on a domain model.

Domains and Subdomains

- Domain can refer to both the entire domain of the business, as well as just one core or supporting area of it.
- The whole Domain of the organization is composed of Subdomains.
- Any attempt to define the business of even a moderately complex organization in a single, all-encompassing model will be at best extremely difficult and will usually fail.
- Almost every software Domain has multiple Subdomains. It really doesn't matter whether the organization is huge and extremely complex or consists of just a few people and the software they use.
- There are different functions that make any business successful, so it's advantageous to think about each of those business functions separately.

Retail

- Customer Relationship Management (CRM)
- Merchandise Management
- Supply Chain Management
- Store Management
- Human Resource Management
- Point of Sale (POS)
- Customer Service
- Marketing
- Accounting

CRM

Customer
- Interests
- Purchasing Power
- Loyalty Points

Supply Chain Management

Recipient
- Address

POS

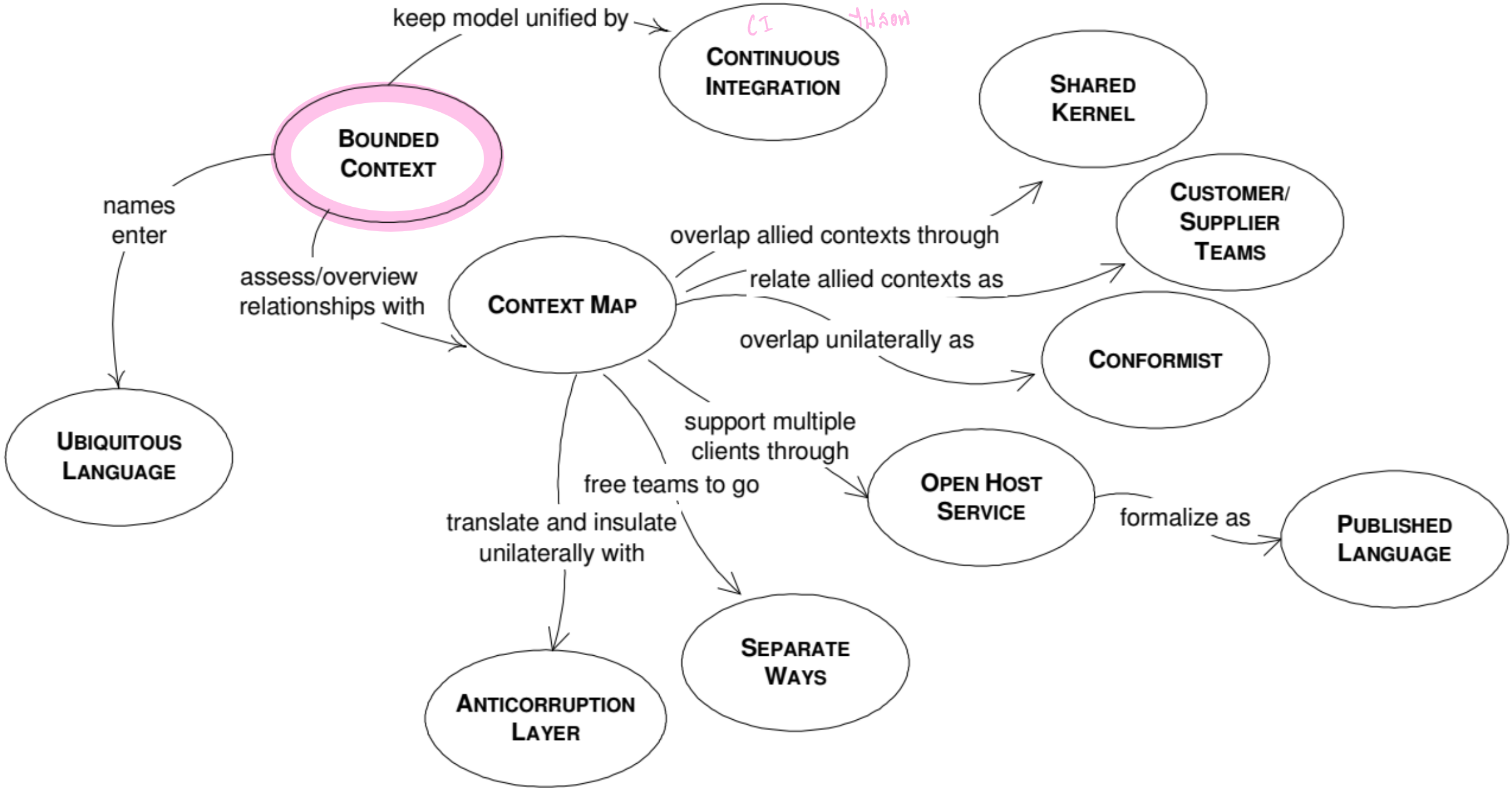
Customer
- Payment Method

Customer Service

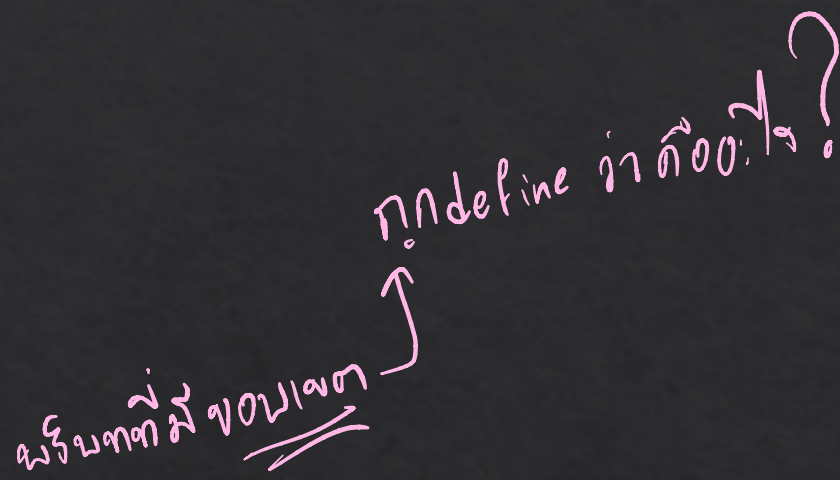
Customer
- Inquiry
- Feedback

Models in Large Software Systems

- The first requirement of a good model is to be consistent, with invariable terms and no contradictions. The internal consistency of a model is called *unification*.
- When the design of the model evolves partially independently, we should consciously divide it into several models.
- Several models well integrated can evolve independently as long as they obey the contract they are bound to.
- Each model should have a clearly delimited border, and the relationships between models should be defined with precision.
- On the next page, we will present a set of techniques used to maintain model integrity and relationships between them.



Marinescu, F., & Avram, A. (2007). *Domain-driven design Quickly*. Lulu. com.



What is a “Bounded Context” ?

“

อธิบายขอบเขต

เช่น sub sys การสั่งของบเขต

A description of a boundary (typically a subsystem, or the work of a particular team) within which a particular model is defined and applicable.

”

The definition of bounded context

Evans, E. (2015). Domain-driven design reference. *Definitions and Pattern Summaries*.

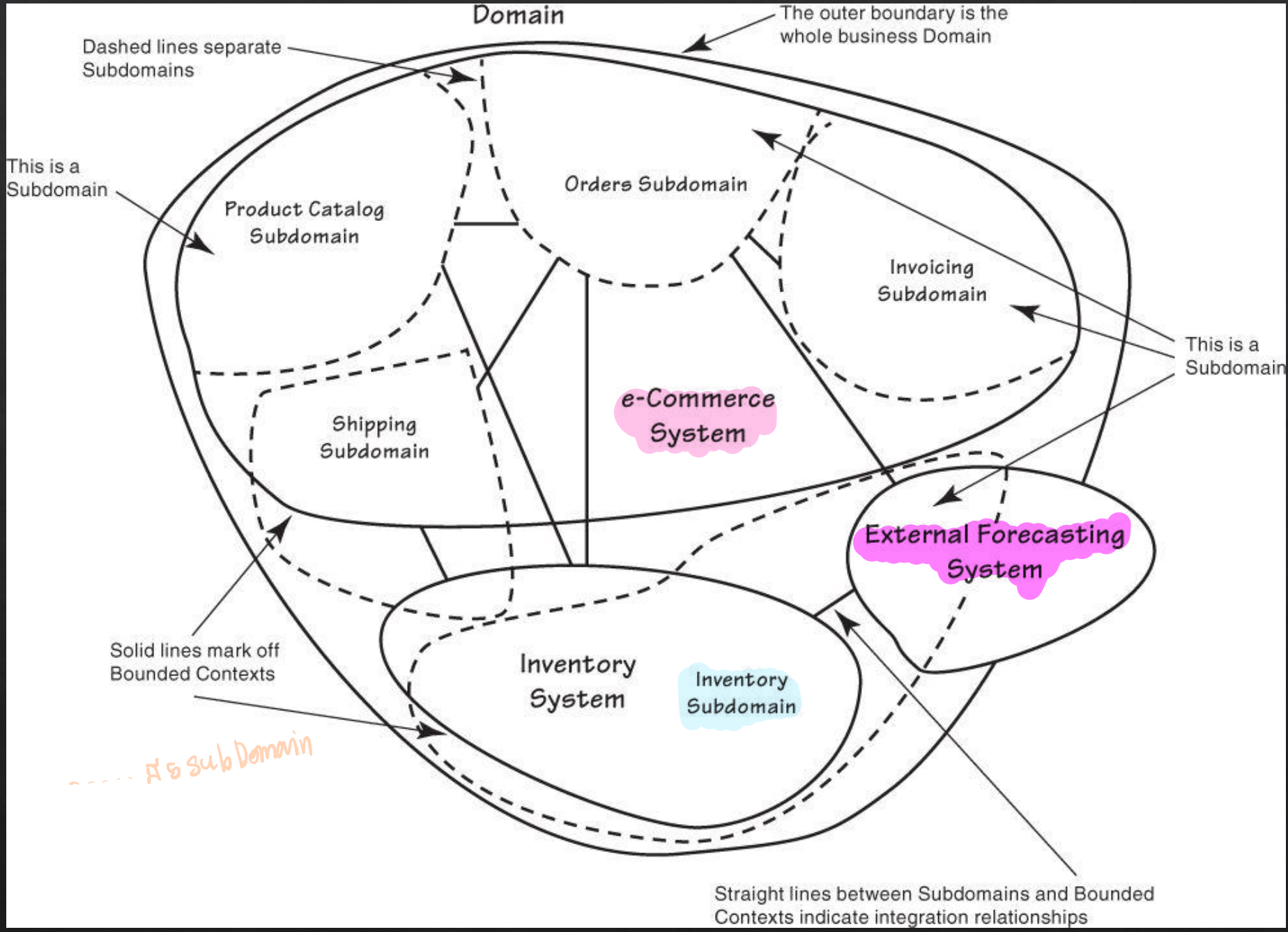
Bounded Contexts in Large Software Systems

- Each model has a context. When we deal with a single model, the context is implicit.
- But when we work on a large enterprise application, we need to define the context for each model we create.
- Multiple models are in play on any large project.
- There is no formula to divide one large model into smaller ones. Try to put in a model those elements which are related, and which form a natural concept.
- *A model should be small enough to be assigned to one team.*
- The main idea is to define the scope of a model, to draw up the boundaries of its context, then do the most possible to keep the model unified.
- Although using DDD to develop a clean Bounded Context is the optimal choice, sometimes we can only wish that had been the case.

Bounded Context vs Module

- A Bounded Context is not a Module.
- A Bounded Context provides the logical frame inside of which the model evolves.
- Modules are used to organize the elements of a model, so Bounded Context encompasses the Module.

Example: E-commerce Application



Vernon, V. (2013). *Implementing domain-driven design*. Addison-Wesley.

Core Domain

- A Core Domain is a part of the business Domain that is of primary importance to the success of the organization.
- Strategically speaking, the business must excel with its Core Domain. It is of utmost importance to the ongoing success of the business.
- That project gets the highest priority, one or more domain experts with deep knowledge of that Subdomain, the best developers, and as much leeway and leverage as possible to give the close-knit team an unobstructed success path.
- Most of your DDD project efforts will be focused on the Core Domain.

Supporting Subdomain & Generic Subdomain

- If it models some aspect of the business that is essential, yet not Core, it is a Supporting Subdomain. The business creates a Supporting Subdomain because it is somewhat specialized.
- Otherwise, if it captures nothing special to the business, yet is required for the overall business solution, it is a Generic Subdomain.
- Being Supporting or Generic doesn't mean unimportant. These kinds of Subdomains are important to the success of the business, yet there is no need for the business to excel in these areas.

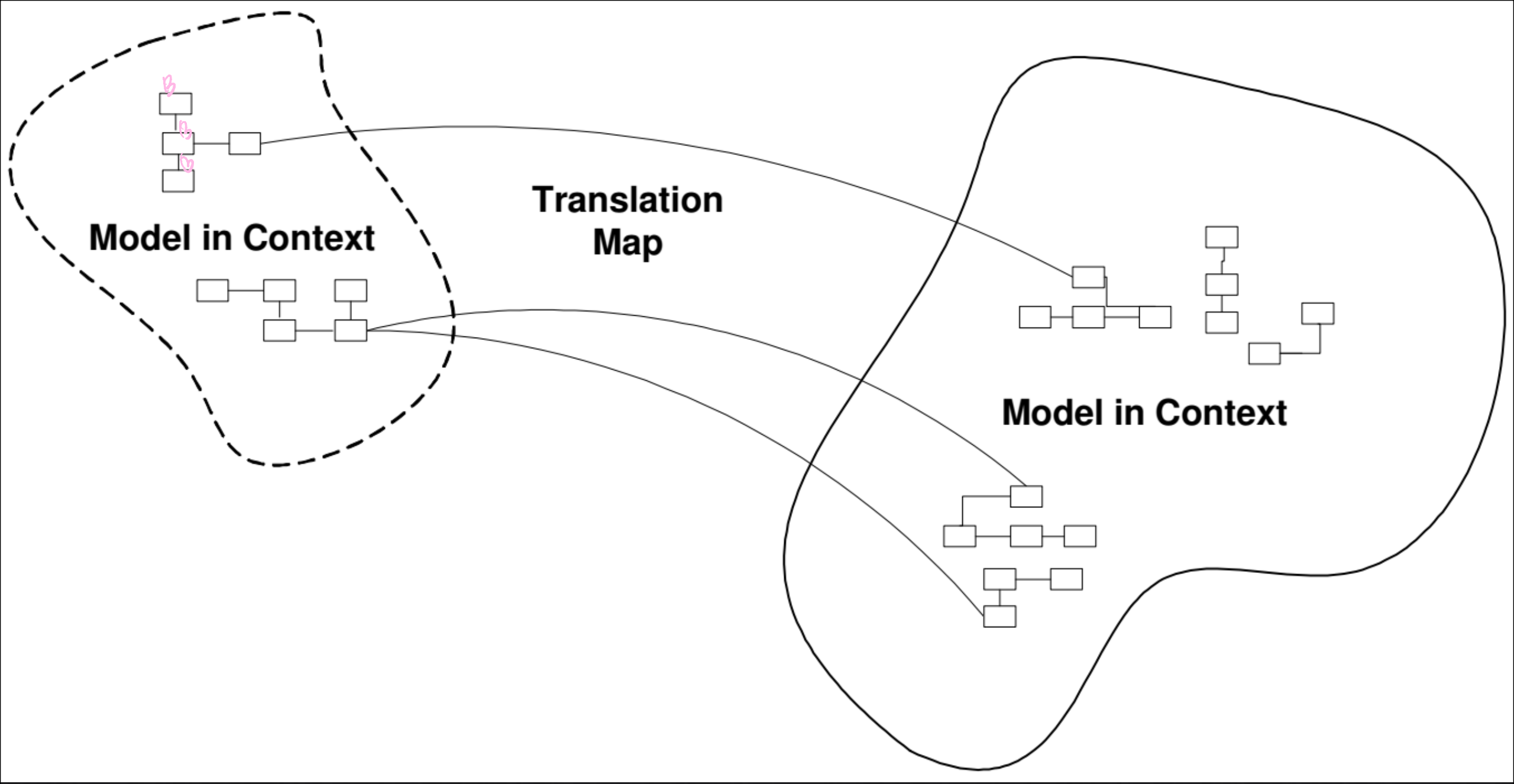
Continuous Integration

- A model is not fully defined from the beginning. It is created, then it evolves continuously based on new insight in the domain and feedback from the development process.
- New concepts may enter the model, and new elements are added to the code.
- All these need are to be integrated into one unified model, and implemented accordingly.
- We need to have a procedure used to merge the code. The sooner we merge the code the better. For a single small team, daily merges are recommended.
- We also need to have a build process in place. The merged code needs to be automatically built so it can be tested.
- Continuous Integration applies to a Bounded Context, it is not used to deal with relationships between neighboring Contexts.

แผนที่ Context → Each B. Context สัมพันธ์กันยังไง

Context Map

- An enterprise application has multiple models, and each model has its own Bounded Context. (แบ่งทีมตาม) (แบ่งทีมตาม)
- It is advisable to use the context as the basis for team organization.
- Each Bounded Context should have a name which should be part of the Ubiquitous Language.
- A Context Map is a document which outlines the different Bounded Contexts and the relationships between them.
- A Context Map can be a diagram like the one on the next page, or it can be any written document. The level of detail may vary.



Marinescu, F., & Avram, A. (2007). *Domain-driven design Quickly*. Lulu. com.

The interaction between different contexts

Partnerships

- When teams in two contexts will succeed or fail together, a cooperative relationship often emerges.

Therefore:

- **Where development failure in either of two contexts would result in delivery failure for both, forge a partnership between the teams in charge of the two contexts.**
- **The teams must cooperate on the evolution of their interfaces to accommodate the development needs of both systems.**
- **Interdependent features should be scheduled so that they are completed for the same release.**
- **It is not necessary, most of the time, for developers to understand the model of the other subsystem in detail, but they must coordinate their project planning.**

Code ร่วมกัน

Shared Kernel

- Sharing a part of the model and associated code is a very intimate interdependency, which can leverage design work or undermine it.

ส่วนสำคัญ ๆ ที่ทีมใช้ร่วมกัน

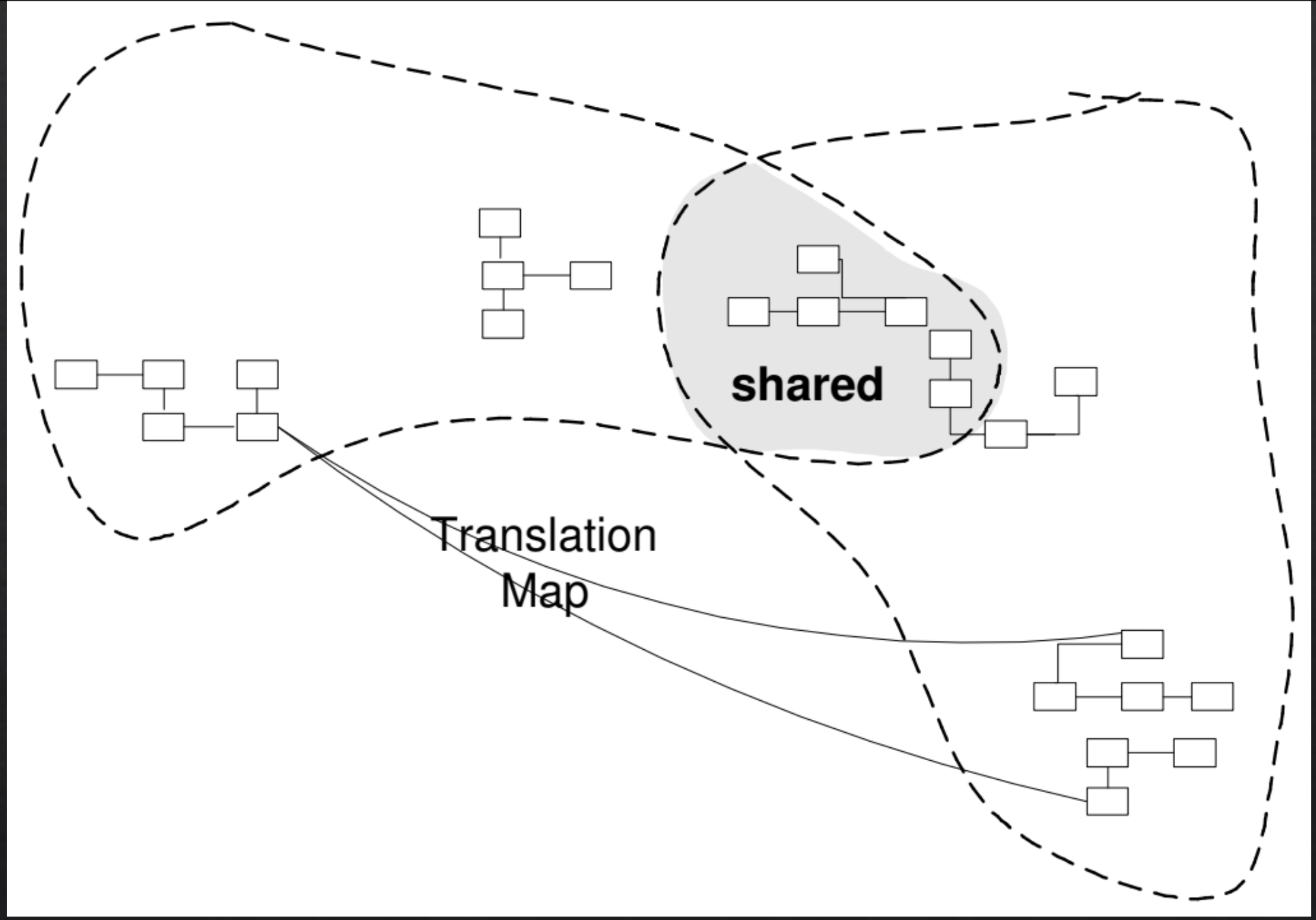
Therefore:

ต้องตกลงร่วมกันให้เล็กที่สุด

- **Designate with an explicit boundary some subset of the domain model that the teams agree to share. Keep this kernel small.**
- **Within this boundary, include, along with this subset of the model, the subset of code or of the database design associated with that part of the model. This explicitly shared stuff has special status, and shouldn't be changed without consultation with the other team.**

ต้องรับรู้ถึงกัน

ถ้าจะเปลี่ยน



Marinescu, F., & Avram, A. (2007). *Domain-driven design Quickly*. Lulu. com.

Customer-Supplier

- When two teams are in an upstream-downstream relationship, where the upstream team may succeed independently of the fate of the downstream team, the needs of the downstream come to be addressed in a variety of ways with a wide range of consequences.

Therefore:

- **Establish a clear customer/supplier relationship between the two teams, meaning downstream priorities factor into upstream planning.**
- **Negotiate and budget tasks for downstream requirements so that everyone understands the commitment and schedule.**

Conformist

- When two development teams have an upstream/down-stream relationship in which the upstream has no motivation to provide for the downstream team's needs, the downstream team is helpless.

Therefore:

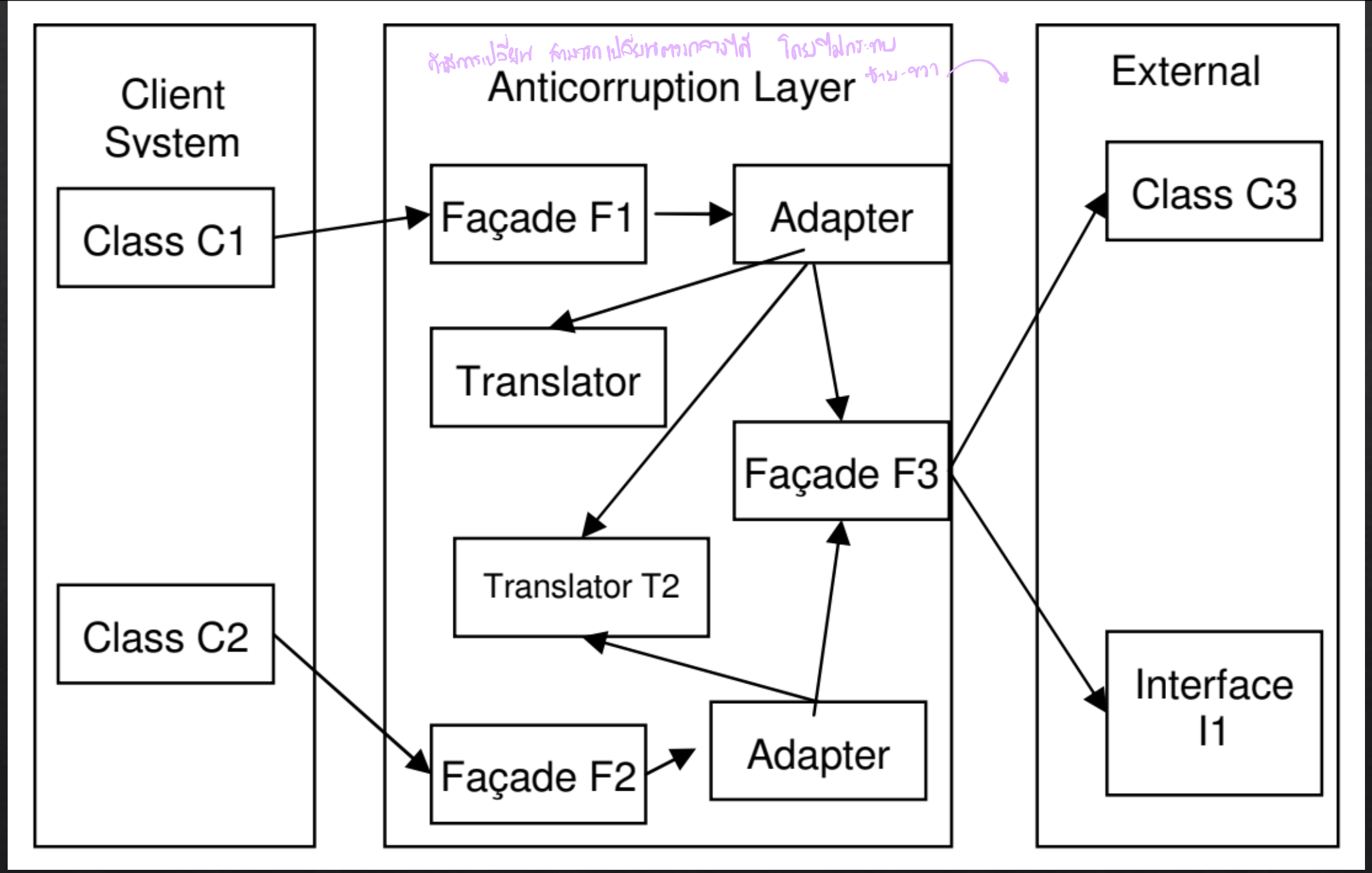
- **Eliminate the complexity of translation between bounded contexts by slavishly adhering to the model of the upstream team.**
- **Also, you will share a ubiquitous language with your upstream team. The upstream is in the driver's seat, so it is good to make communication easy for them. Altruism may be sufficient to get them to share information with you.**

Anticorruption Layer

- Translation layers can be simple, even elegant, when bridging well-designed bounded contexts with cooperative teams.
- But when control or communication is not adequate to pull off a shared kernel, partner or customer/supplier relationship, translation becomes more complex. The translation layer takes on a more defensive tone.

Therefore:

- **As a downstream client, create an isolating layer to provide your system with functionality of the upstream system in terms of your own domain model. This layer talks to the other system through its existing interface, requiring little or no modification to the other system.**
- **Internally, the layer translates in one or both directions as necessary between the two models.**



Marinescu, F., & Avram, A. (2007). *Domain-driven design Quickly*. Lulu. com.

Open-host Service

- Typically for each bounded context, you will define a translation layer for each component with which you have to integrate that is outside the context.
- Where integration is one-off, this approach of inserting a translation layer for each external system avoids corruption of the models with a minimum of cost.

Therefore:

- **When you find your subsystem in high demand, you may need to define a protocol that gives access to your subsystem as a set of services. Open the protocol so that all who need to integrate with you can use it.**
- **Enhance and expand the protocol to handle new integration requirements, except when a single team has idiosyncratic needs.**
- **Then, use a one-off translator to augment the protocol for that special case so that the shared protocol can stay simple and coherent.**

Published Language

- The translation between the models of two bounded contexts requires a common language.

Therefore:

- **Use a well-documented shared language that can express the necessary domain information as a common medium of communication, translating as necessary into and out of that language.**

Separate Ways

- If two sets of functionality have no significant relationship, they can be completely cut loose from each other.

Therefore:

- **Declare a bounded context to have no connection to the others at all, allowing developers to find simple, specialized solutions within this small scope.**

Big Ball of Mud

- As we survey existing software systems, trying to understand how distinct models are being applied within defined boundaries, we find parts of systems, often large ones, where models are mixed and boundaries are inconsistent.
- When well-defined context boundaries are absent, or disappear, multiple conceptual systems and mix together, making definitions and rules ambiguous or contradictory.
- The systems are made to work by contingent logic as features are added.
- Eventually the software congeals into a big ball of mud.

Therefore:

- **Draw a boundary around the entire mess and designate it a big ball of mud. Do not try to apply sophisticated modeling within this context. Be alert to the tendency for such systems to sprawl into other contexts.**

Distillation

Distillation

- Distillation is the process of separating the substances composing a mixture.
- The purpose of distillation is to extract a particular substance from the mixture. During the distillation process, some byproducts may be obtained, and they can also be of interest.
- A large domain has a large model even after we have refined it and created many abstractions. It can remain big even after many refactorings.
- In situations like this, it may be time for a distillation. The idea is to define a Core Domain which represents the essence of the domain.
- The byproducts of the distillation process will be Generic Subdomains which will comprise the other parts of the domain.

Core Domain

- In a large system, there are so many contributing components, all complicated and all absolutely necessary to success, that the essence of the domain model, the real business asset, can be obscured and neglected.
- It is harsh reality that not all parts of the design are going to be equally refined. Priorities must be set.

Therefore:

- **Boil the model down. Define a core domain and provide a means of easily distinguishing it from the mass of supporting model and code.**
- **Make the core small.**
- **Apply top talent to the core domain, and recruit accordingly.**

Generic Subdomains

- Some parts of the model add complexity without capturing or communicating specialized knowledge. Anything extraneous makes the core domain harder to discern and understand.
- Yet, however generic, these other elements are essential to the functioning of the system and the full expression of the model.

Therefore:

- **Identify cohesive subdomains that are not the motivation for your project.**
- **Factor out generic models of these subdomains and place them in separate modules.**
- **Once they have been separated, give their continuing development lower priority than the core domain, and avoid assigning your core developers to the tasks.**
- **Also consider off-the-shelf solutions or published models for these generic subdomains.**

Domain Vision Statement

- At the beginning of a project, the model usually doesn't even exist, yet the need to focus its development is already there.
- In later stages of development, there is a need for an explanation of the value of the system that does not require an in-depth study of the model.

Therefore:

- Write a short description (about one page) of the core domain and the value it will bring, the "value proposition."
- Ignore those aspects that do not distinguish this domain model from others.
- Show how the domain model serves and balances diverse interests. Keep it narrow.
- Write this statement early and revise it as you gain new insight.

Highlighted Core

- A domain vision statement identifies the core domain in broad terms, but it leaves the identification of the specific core model elements up to the vagaries of individual interpretation.
- Unless there is an exceptionally high level of communication on the team, the vision statement alone will have little impact.

Therefore (as one form of highlighted core):

- **Write a very brief document (three to seven sparse pages) that describes the core domain and the primary interactions among core elements.**

and/or (as another form of highlighted core):

- **Flag the elements of the core domain within the primary repository of the model. Make it effortless for a developer to know what is in or out of the core.**

- Although the vision statement and highlighted core inform and guide, they do not actually modify the model or the code itself.

- Partitioning generic subdomains physically removes some distracting elements.

- Next, we'll look at other ways to structurally change the model and the design itself to make the core domain more visible and manageable.

Cohesive Mechanisms

- Computations sometimes reach a level of complexity that begins to bloat the design. The conceptual "what" is swamped by the mechanistic "how." A large number of methods that provide algorithms for resolving the problem obscure the methods that express the problem.

Therefore:

- **Partition a conceptually cohesive mechanism into a separate lightweight framework. Now the other elements of the domain can focus on expressing the problem ("what"), delegating the intricacies of the solution ("how") to the framework.**
- **Factoring out generic subdomains reduces clutter, and cohesive mechanisms serve to encapsulate complex operations.**

Segregated Core

- Elements in the model may partially serve the core domain and partially play supporting roles. Core elements may be tightly coupled to generic ones. The conceptual cohesion of the core may not be strong or visible. All this clutter and entanglement chokes the core.
- Designers can't clearly see the most important relationships, leading to a weak design.

Therefore:

- **Refactor the model to separate the core concepts from supporting players (including ill defined ones) and strengthen the cohesion of the core while reducing its coupling to other code.**
- **Factor all generic or supporting elements into other objects and place them into other packages, even if this means refactoring the model in ways that separate highly coupled elements.**

Abstract Core

- When there is a lot of interaction between subdomains in separate modules, either many references will have to be created between modules, which defeats much of the value of the partitioning, or the interaction will have to be made indirect, which makes the model obscure.

Therefore:

- **Identify the most fundamental differentiating concepts in the model and factor them into distinct classes, abstract classes, or interfaces.**
- **Design this abstract model so that it expresses most of the interaction between significant components.**
- **Place this abstract overall model in its own module, while the specialized, detailed implementation classes are left in their own modules defined by subdomain.**

Recommended Resources

- Domains, Sub-Domains and Bounded Contexts: Explained with example from industry – Alok Mishra (alok-mishra.com)
- ออกแบบ Microservices ด้วย Domain Driven Design #1 - Concept Overview – YouTube
- Bounded Contexts - Eric Evans - DDD Europe 2020 – YouTube
- The Art of Discovering Bounded Contexts by Nick Tune - YouTube
- Bounded Contexts, Microservices, and Everything In Between - Vladik Khononov - KanDDDinsky 2018 – YouTube
- BoundedContext (martinfowler.com)