sakonnet technology
energy trading and risk systems

Re-architecting a Large Code Base

The "Remodularization" of Xenon

Timothy High (timothy.high@sknt.com)
Software Architect
Sakonnet Technologies (www.sknt.com)

Ian Sutton (ian.sutton@headwaysoftware.com)
Chief Architect
Headway Software (www.headwaysoftware.com)

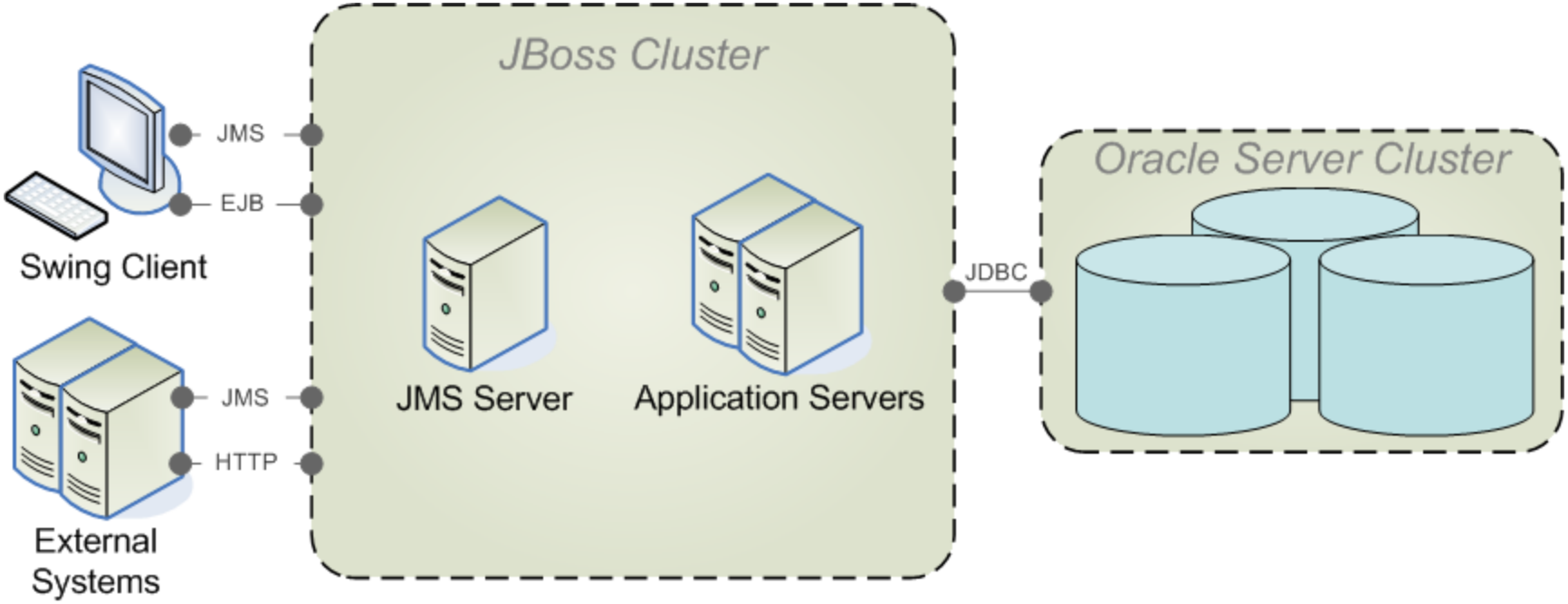
Sakonnet Company Profile

- SaaS leader in the ETRM space
- ~35 developers
  - Java, PL/SQL
  - All experienced or senior
  - All architects are “hands-on”
- Distributed team (NY, Rio, Germany)
- Not afraid of changes
  - Award-winning migration from Websphere to JBoss
  - ObjectStore to Oracle migration

About Xenon

- Software Characteristics
  - Same code base since 1999
  - Java & PL/SQL
  - JBoss/J2EE/Spring/Oracle
  - 5600 public classes, 738 KLOC (Java only)
  - Strong domain layer (a Martin Fowler-style *Domain Model*)
  - Persistence: ObjectStore (OODBMS) → Hibernate + Oracle

System Architecture



Remodularization

The Problem:

- The Xenon code base had outgrown its initial design
- Interdependencies were becoming a burden
- Developers weren't always sure where to put new code
- The GUI Client JARs were bloated
- Build times were over an hour

The Solution:

- Define a new standard for Xenon modules
- Define a new overall architecture
- Refactor existing code to the new structure

Refactoring: to make changes to a body of code in order to improve its internal structure, without changing its external behavior.

What is a Module?

A cohesive unit

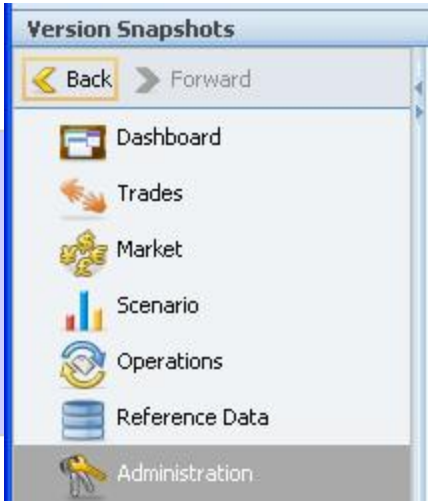
- Libraries
- Business functionality
- Infrastructure services
- A component

In Xenon:

Functional Module

Business concept

- Trades
- Market Data
- Reports



Architectural Module

A Maven project

- Domain
- GUI
- Lib

What is a Module?

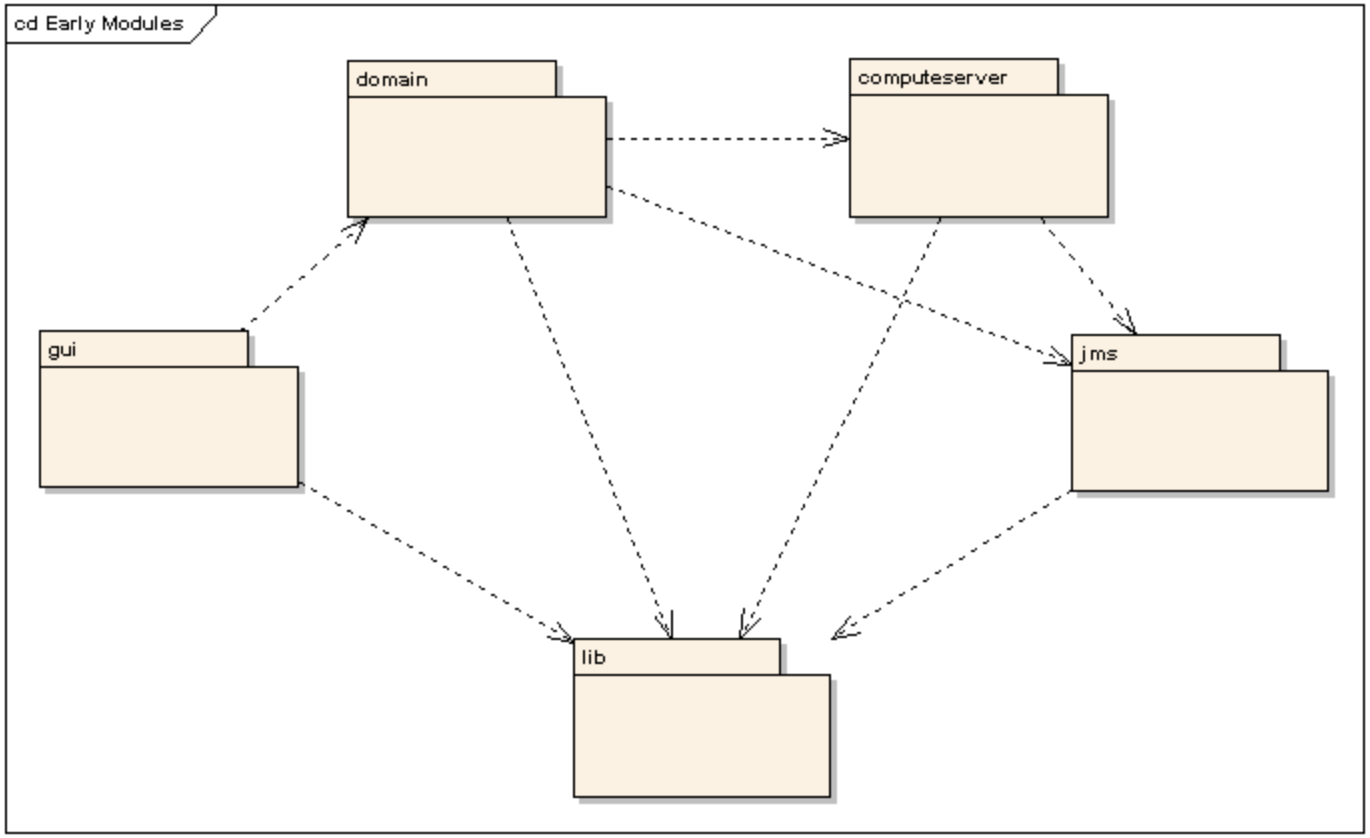
Definitions from *Documenting Software Architecture: Views and Beyond (Clements, Bachmann, Bass, et. al.)*

- An implementation unit that provides a coherent unit of functionality
  - Mostly design-time entity
  - Static element (as opposed to "component")
- Purpose
  - Decomposition of a whole (reduce complexity)
  - Encapsulation
  - Information hiding
- Characteristics
  - Name (with possible namespaces)
  - Interface/list of responsibilities
  - Relationships: is part of (decomposition), uses, allowed to use, inherits from

→ *For our purposes, a module is a Maven project*

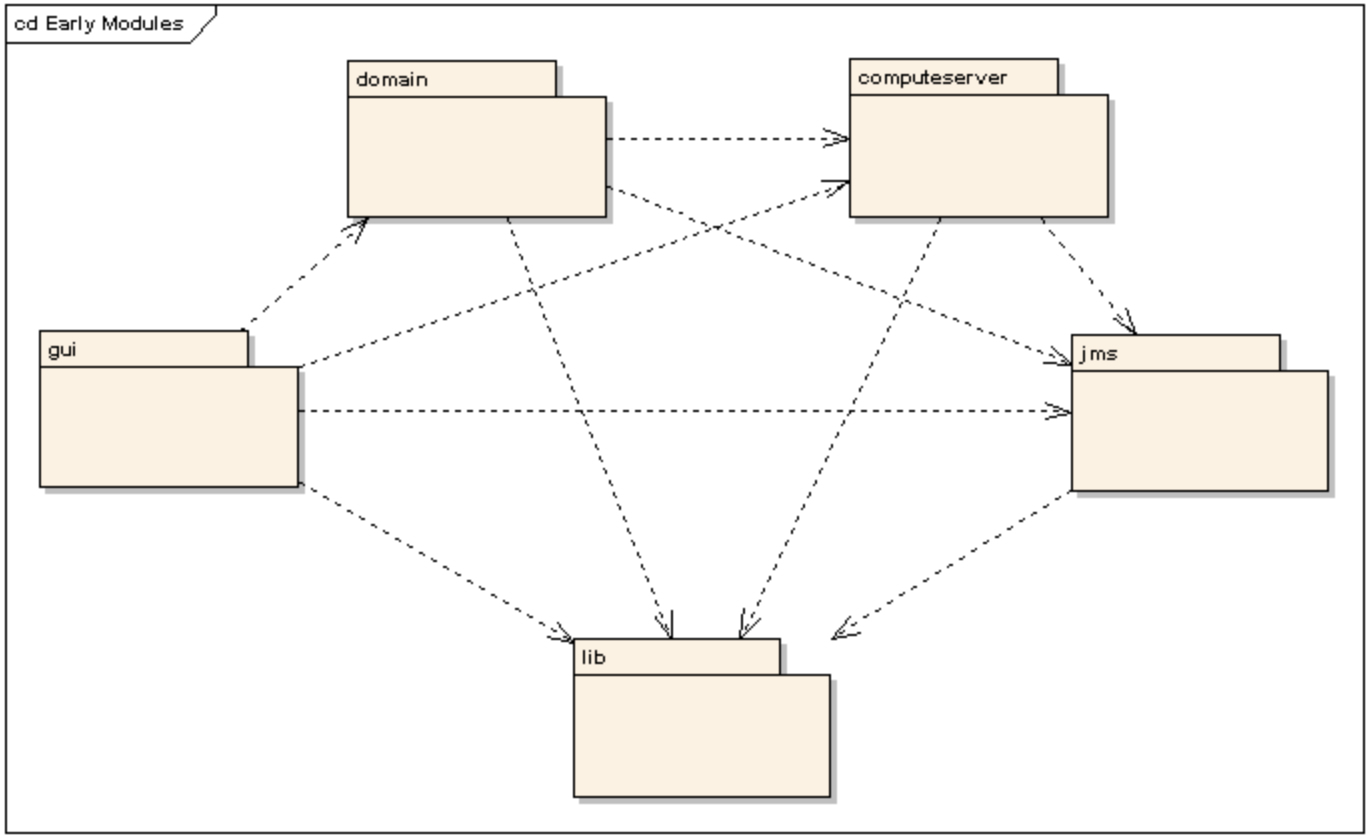
Xenon Modules: A History

In there beginning there were a few clearly-defined modules



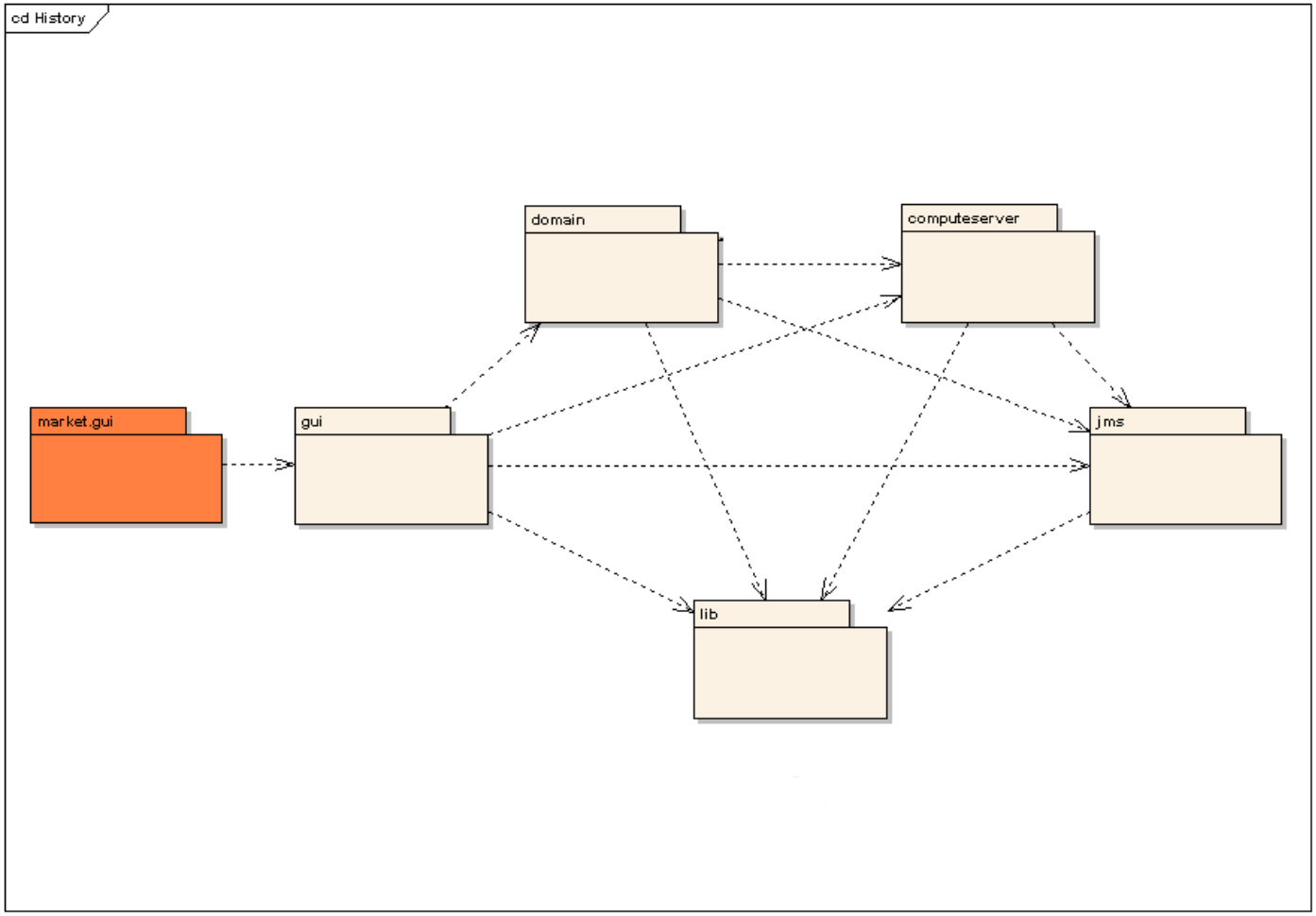
Xenon Modules: A History

As the application grew, so did the dependencies



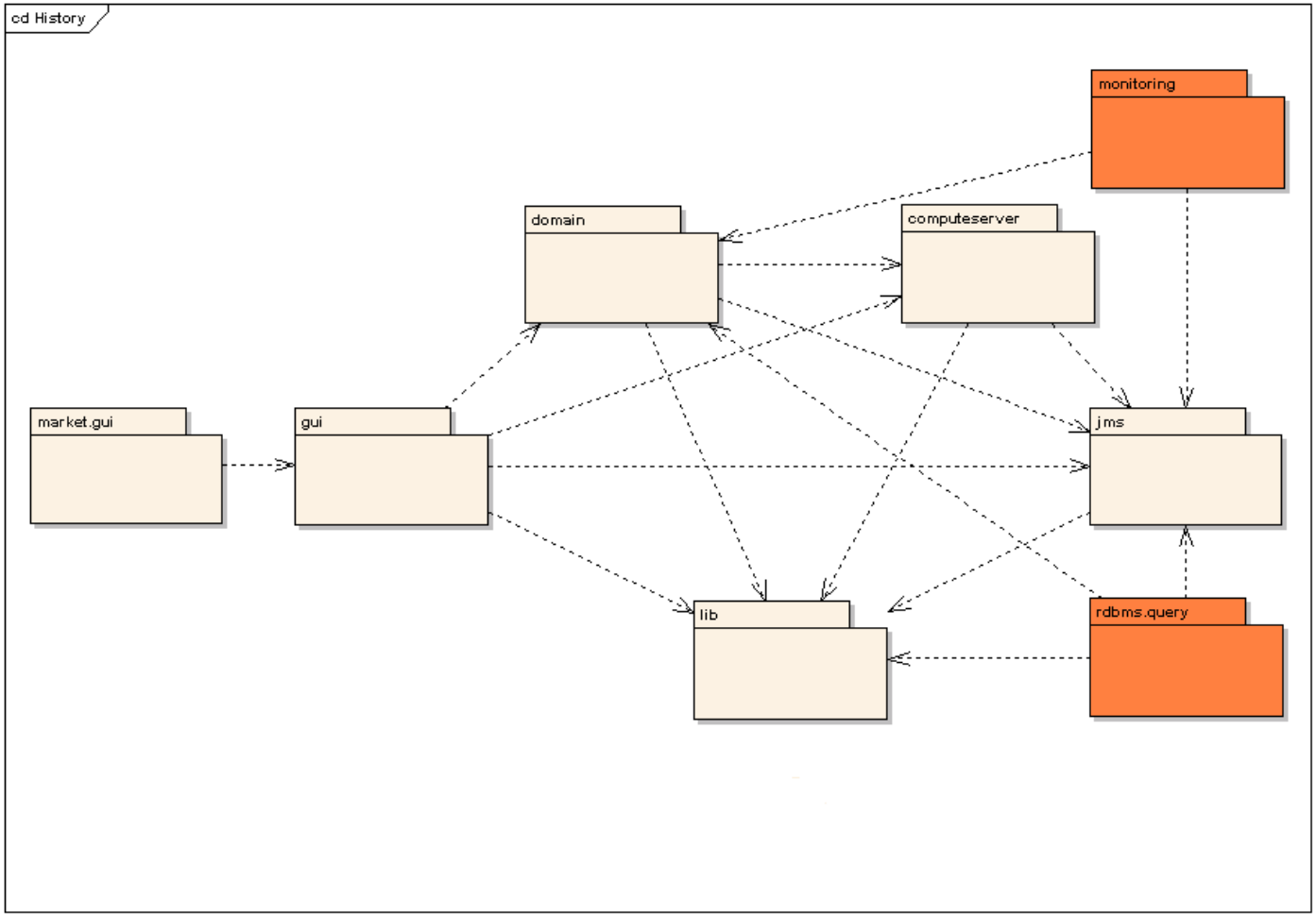
Xenon Modules: A History

New functionality sometimes brought new modules



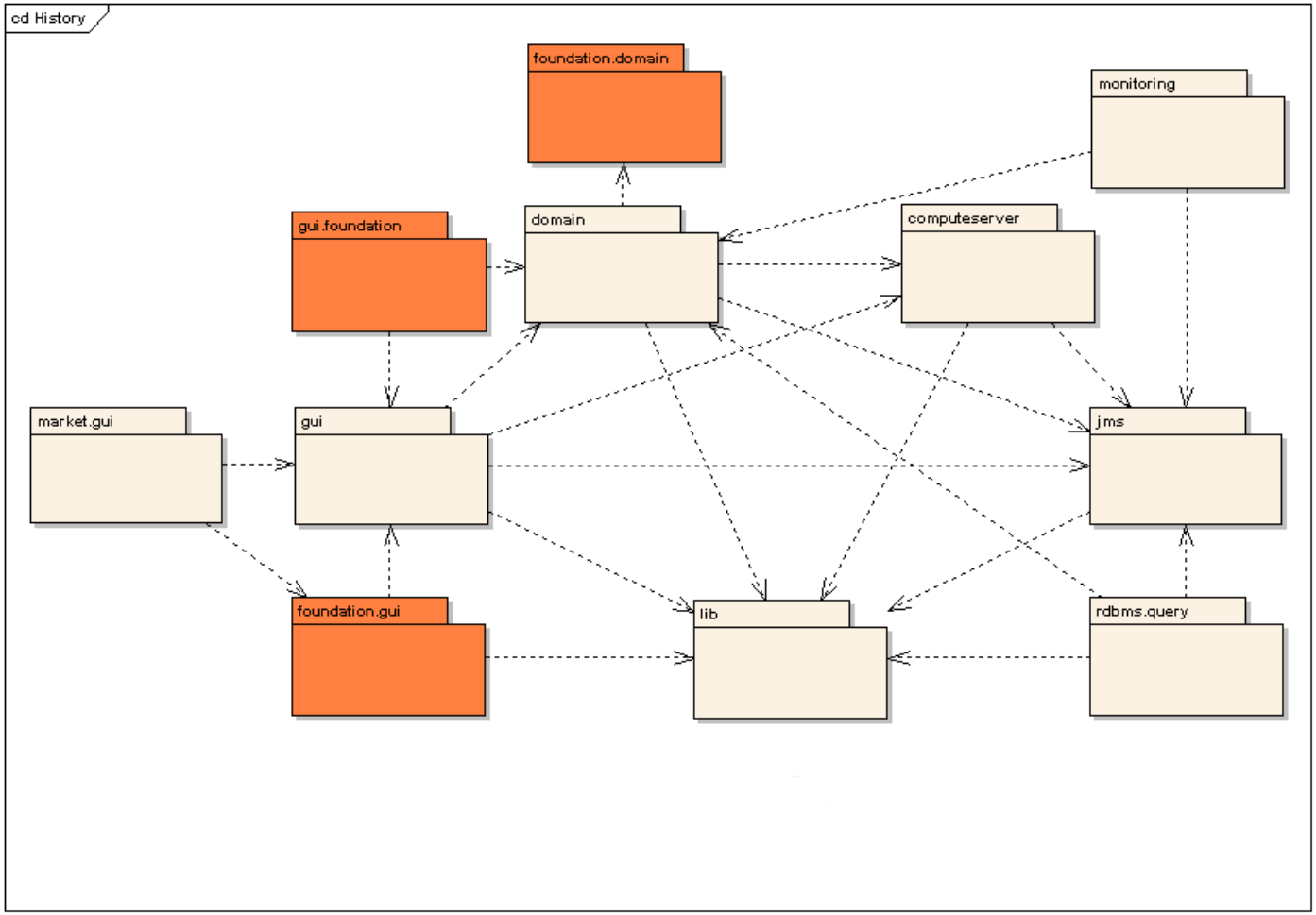
Xenon Modules: A History

Performance and quality of service enhancements left their marks



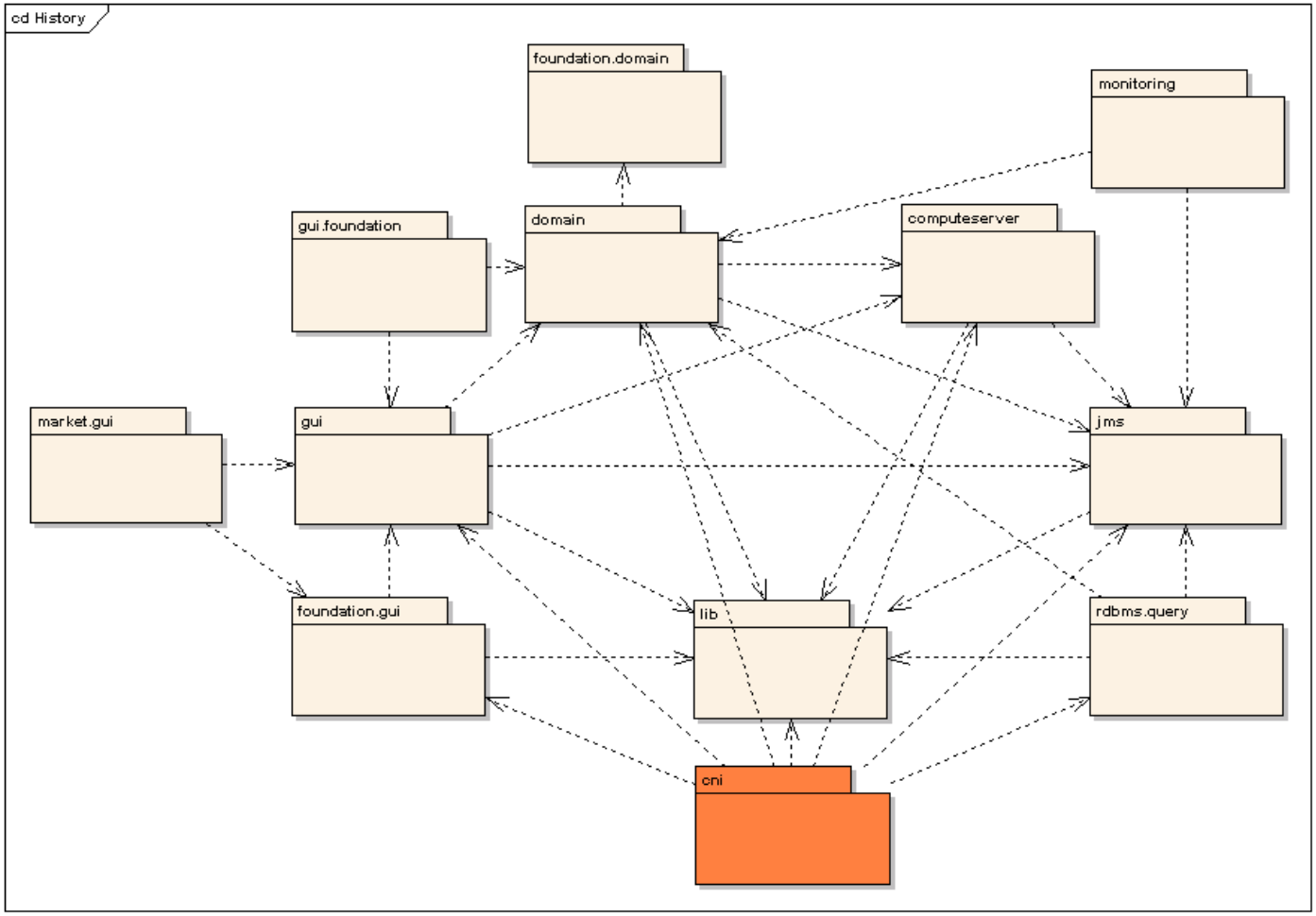
Xenon Modules: A History

Efforts were made to improve the design and increase reusability



Xenon Modules: A History

And sometimes, entire functional units were rewritten



Causes of Decay

"Copy-paste Architecture" (aka "Architecture by Accident"):

- Focus not on long-term impacts
- May have many conflicting conventions
  - "Magic 8-Ball" decisions

"Big Ball of Mud" design patterns:

- It may not be pretty, but it *works*
- http://www.laputan.org/mud/

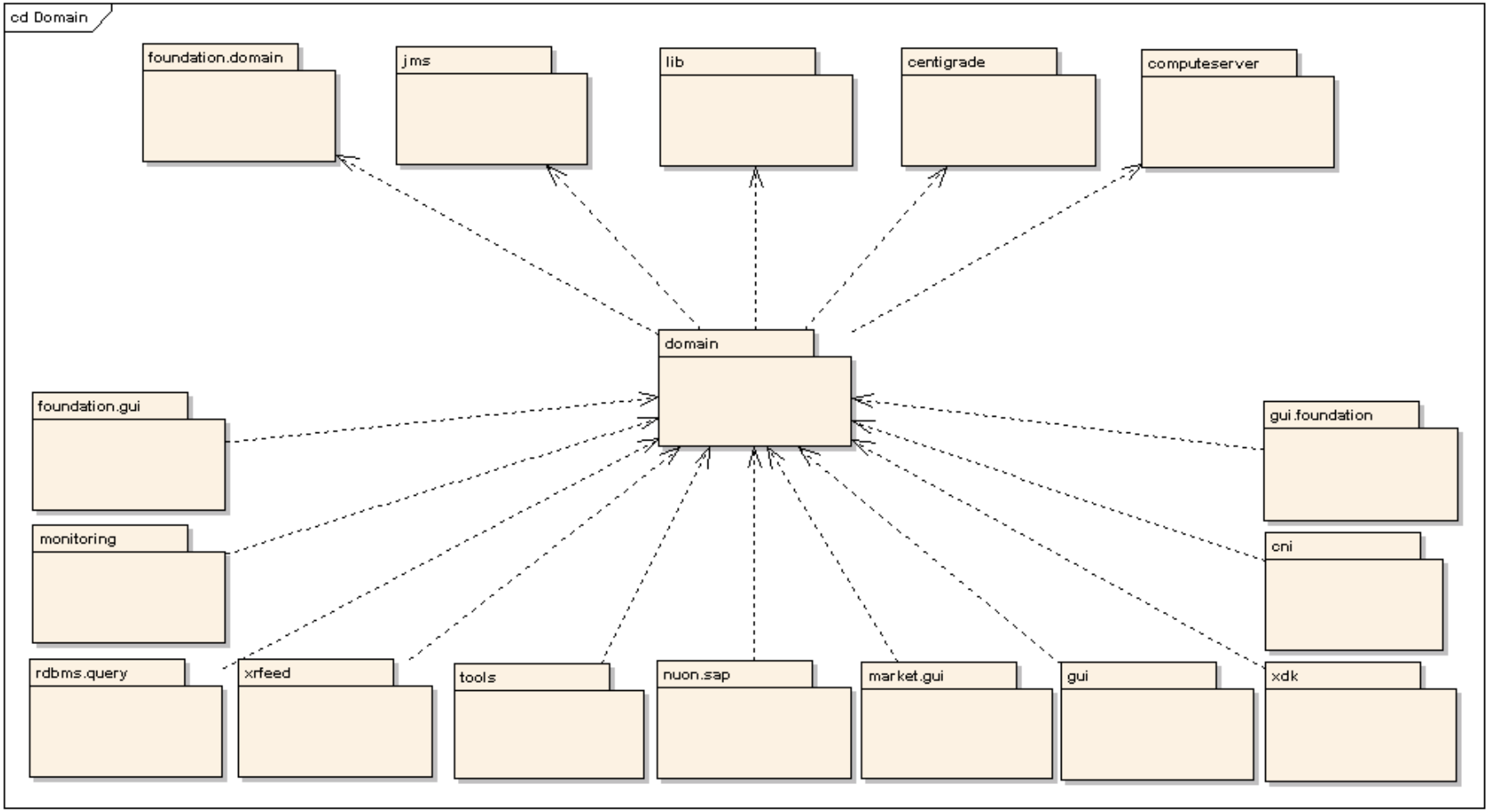
Causes of Decay

- Wrong reasons to create a module:
  - Created for a project (too transient)
  - "False start" redesign (refactoring without analysis and standardization)
  - Not sure which module should host new code
- "Allowed-to-use" = will use
  - Java enforcement of layered architecture limited at best
- Invisible architecture
  - Can't control what you can't see

- Xenon-specific issues:
  - ObjectStore created inertia against refactoring
  - Reuse of strong Domain Model in client led to GUI-specific concerns in model

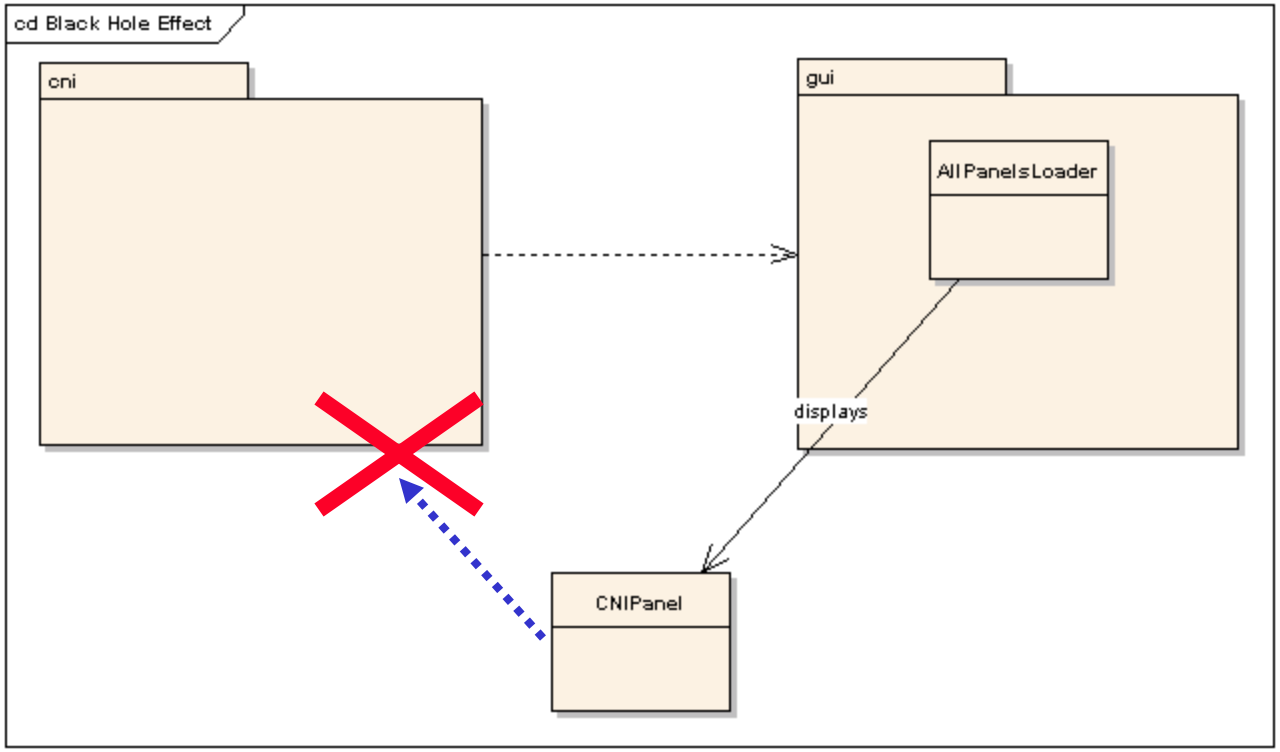
Signs of Decay

Due to dependencies, a small code change could have a large impact



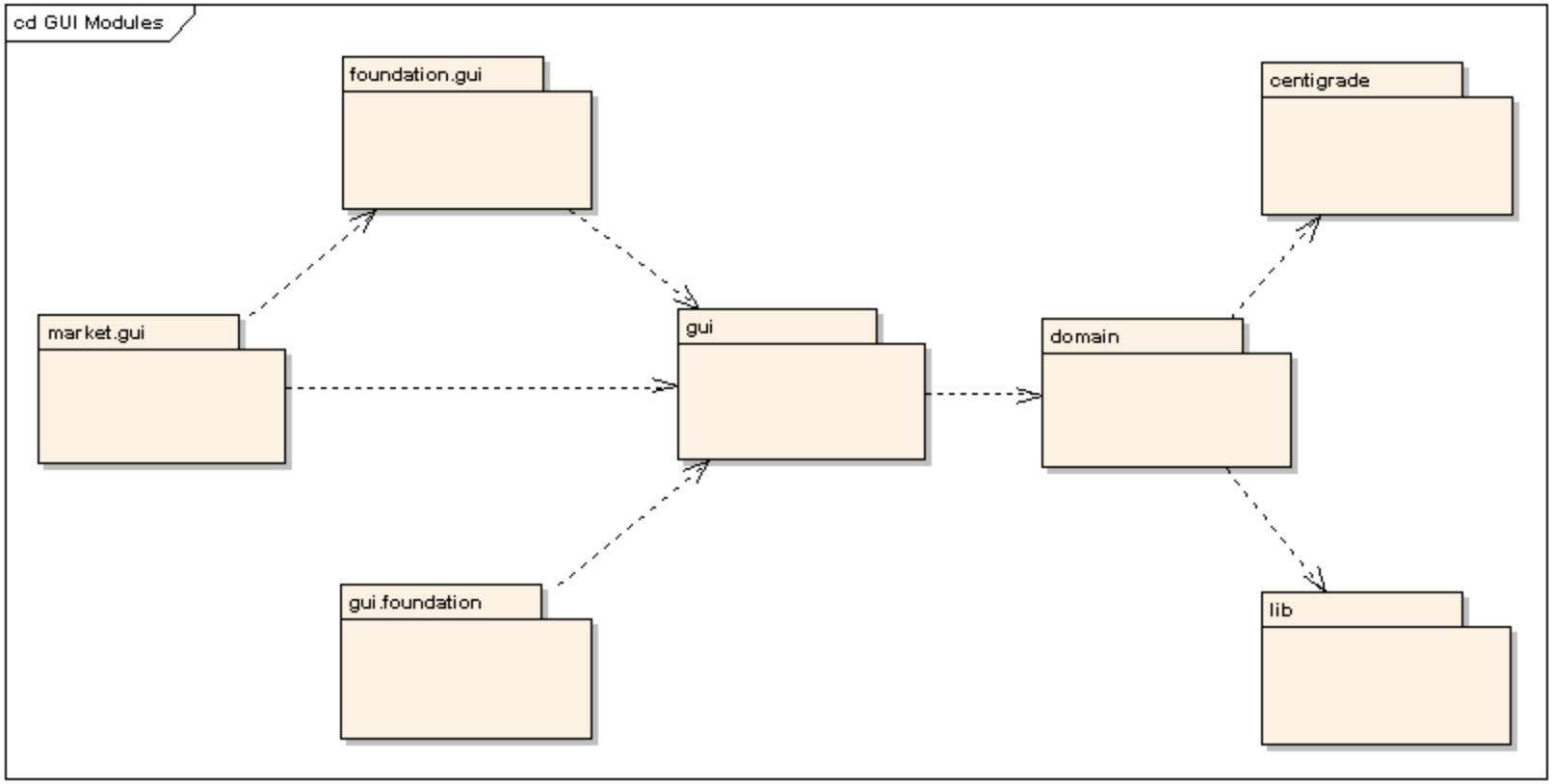
Signs of Decay

Black Hole Effect: It is not always possible to place code where it “belongs”



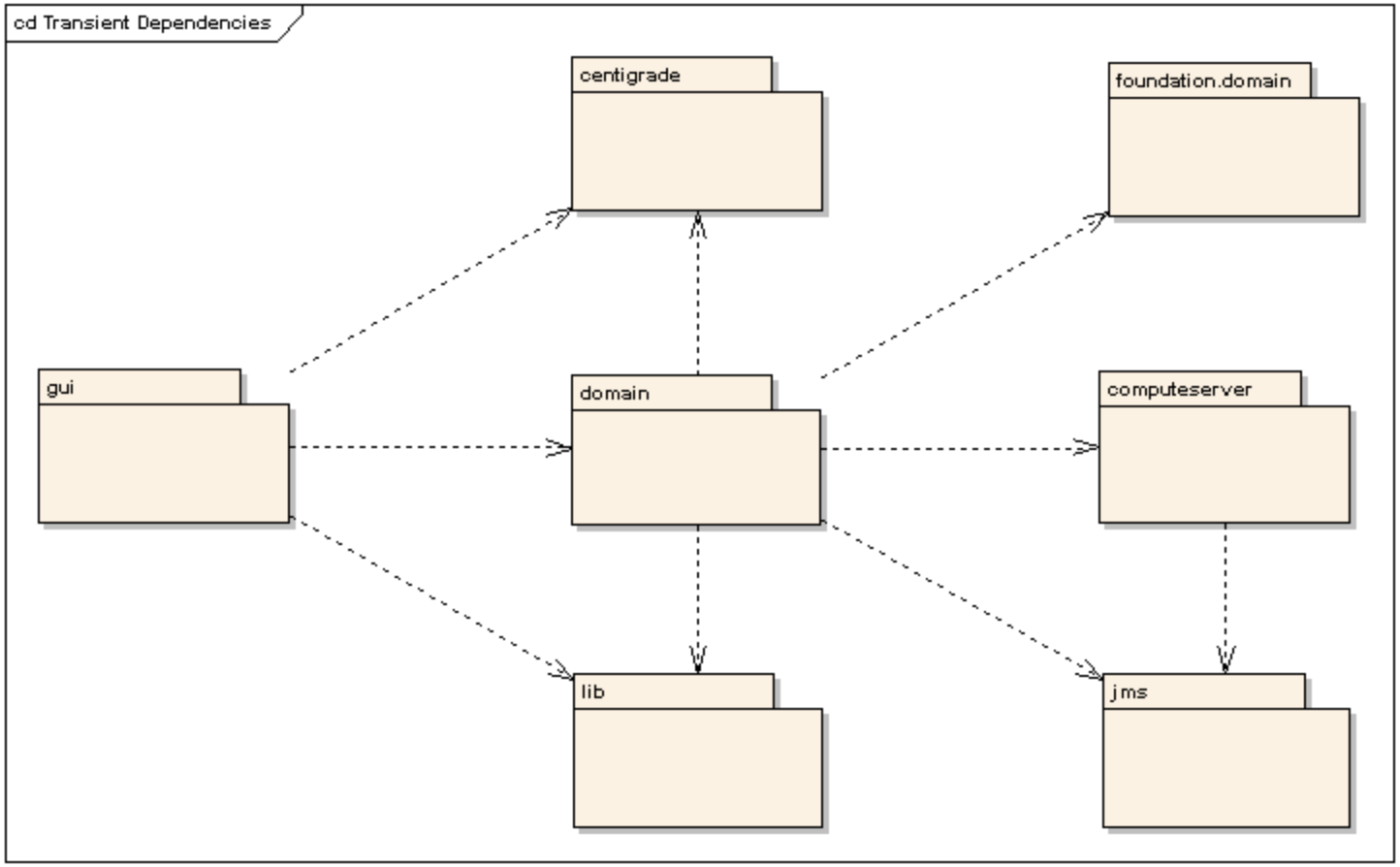
Signs of Decay

Proliferation of undefined special-purpose modules can cause confusion



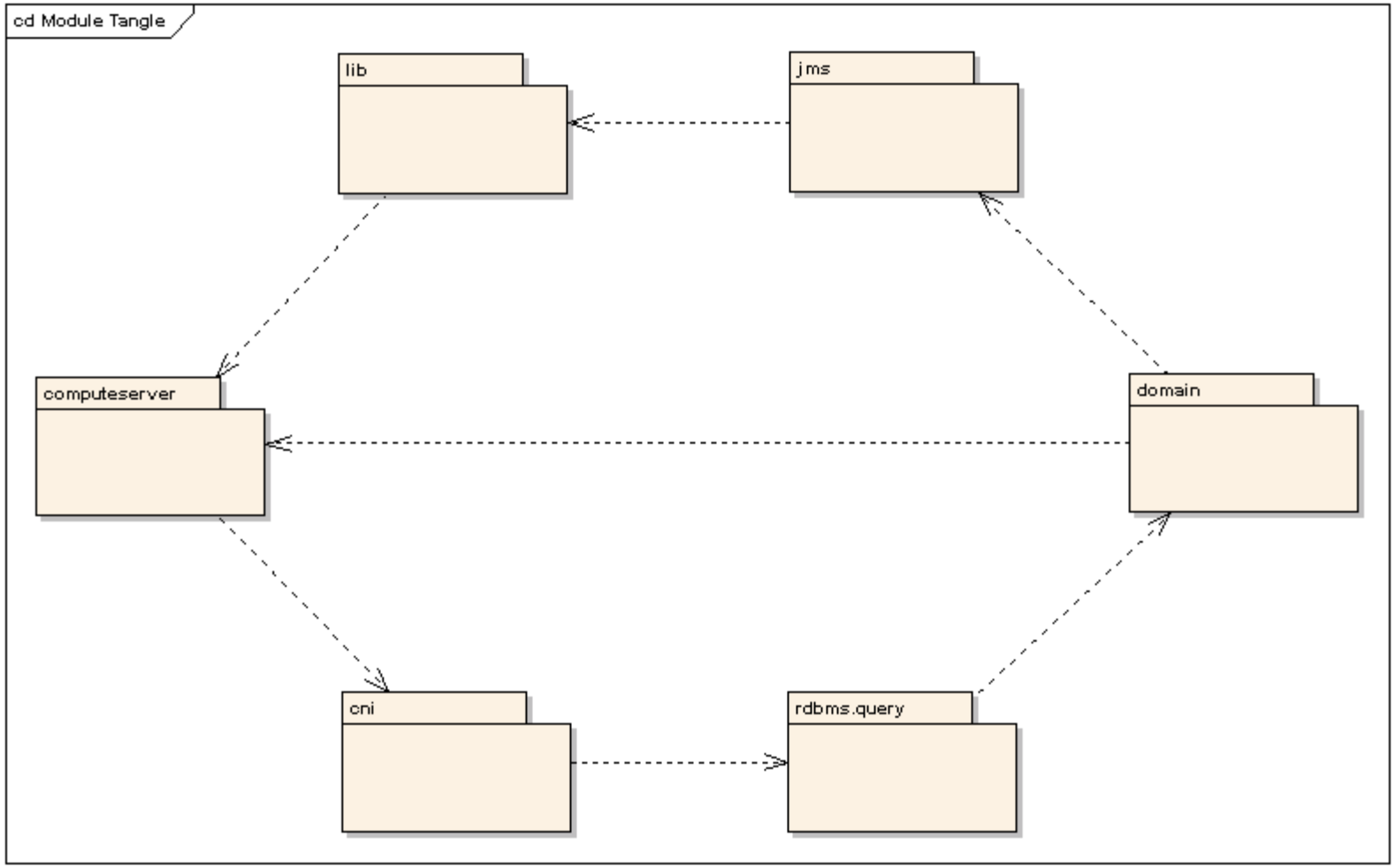
Signs of Decay

Coarse-grained modules force deployment of unnecessary code



Signs of Decay

Cyclic dependencies may force rebuilds and testing of unchanged code



Problems with Packages

- Package name didn't indicate module membership
  - A consequence of ObjectStore and a previous attempt at remodularization
- No consistent standard
- Many packages didn't reflect a layered structure
- Developers were unsure where to put new code
- Too many classes in a single package
  - Rather than make another arbitrary decision, easier to use existing package
- Lack of Java support to enforce layering
  - Reverse and cyclic dependencies common

The Solution: Remodularization

Objectives:

- Define a clear and simple overall architecture
- Provide a clear standard for future development
- Reduce build times
- Reduce maintenance time / increase productivity
- Improve application flexibility

Challenges

- Hard to measure business value
  - Project was already “sold”, but wanted to track benefits
  - Hard to measure opportunity costs
- Have to do it during ongoing development
  - Can’t stop the train
  - Can’t leave anything half-working
  - Developers can’t become “disoriented”
  - How to do it once and make sure we don’t backtrack?
  - Very limited resources (1 dev + team lead + architect)
- Multiple branches
  - Refactoring in trunk complicates merges from other branches

Project Strategy

3 phases:

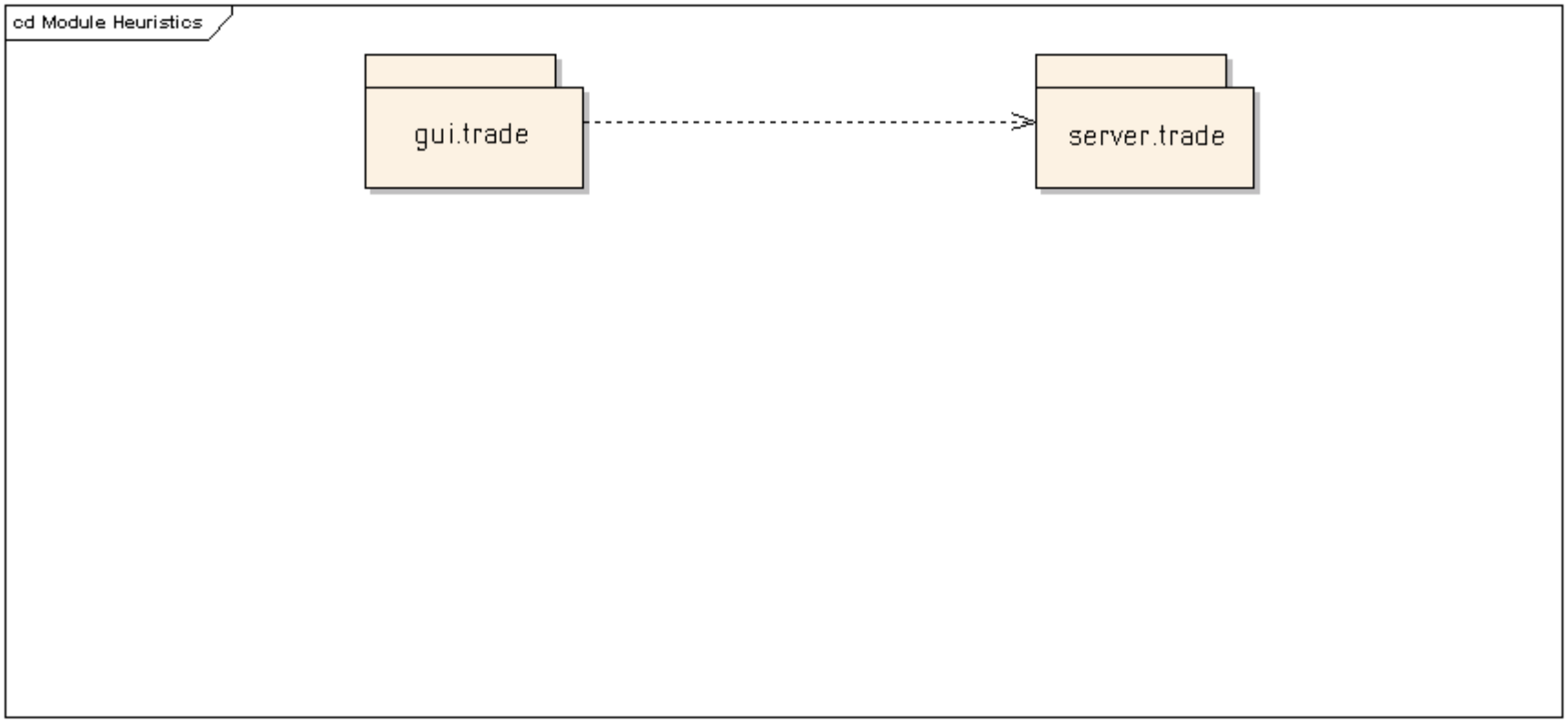
- Analysis (1 month)
  - Structure101
- Tools and Preparation (2 months)
  - Maven 2
  - Hudson
  - Structure101
  - IDE support
- Active Remodularization (ongoing)
  - Structure101
  - Eclipse/IntelliJ
  - Subversion (can’t lose history!)

Phase I: Analysis & Planning

- Produce first draft of module standards
- Objectives, risk analysis, project strategy
- Top-down:
  - General principles for modules
  - How to achieve objectives
- Bottom-up:
  - Structure101:
    - Analysis of existing dependencies
    - Design of new modules
    - “Simulation” of refactoring
  - Work in cooperation with Team Leads & members
  - Not all violations can be resolved by moving classes
    - Remaining Structure101 layering violations require code changes

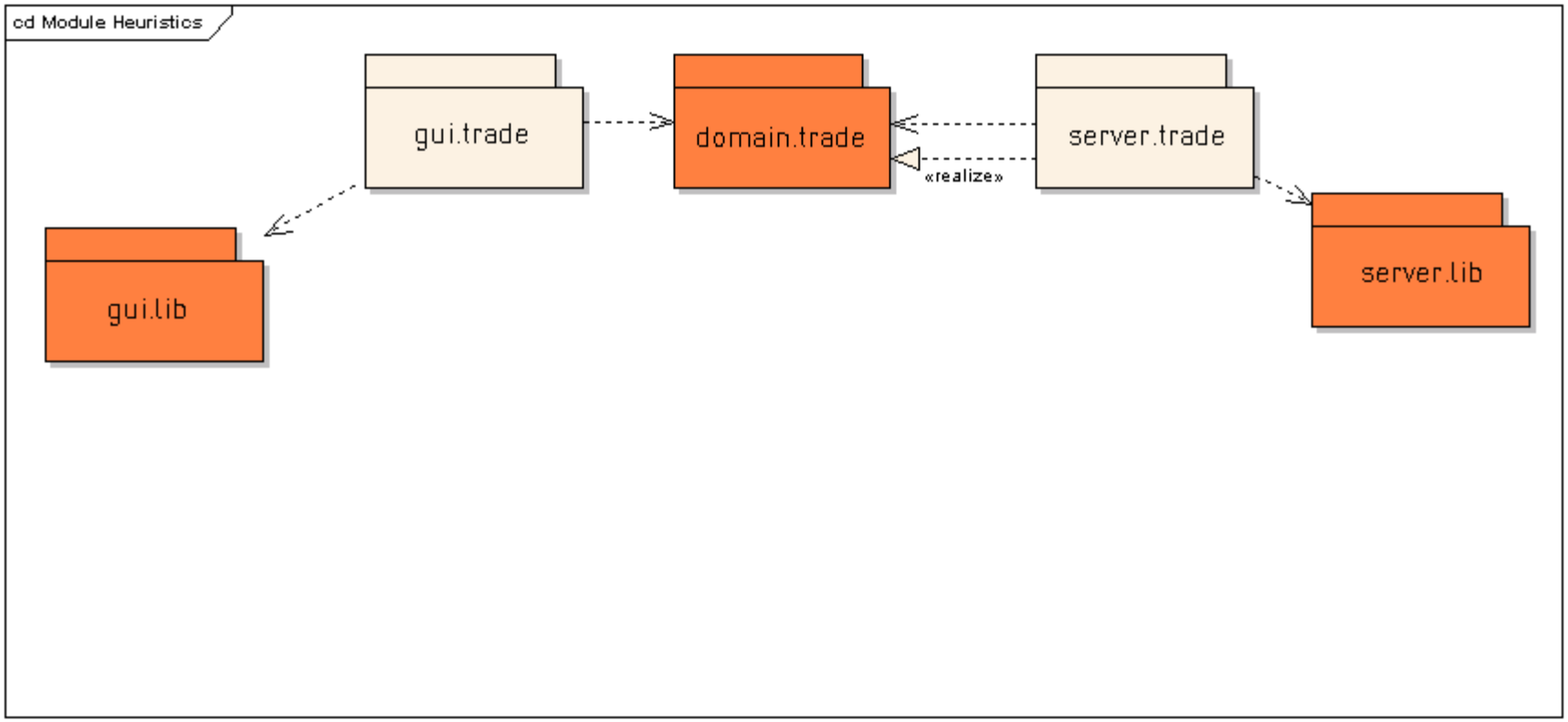
Signs You May Need a New Module

Deployment: different code groupings located in separate nodes or tiers



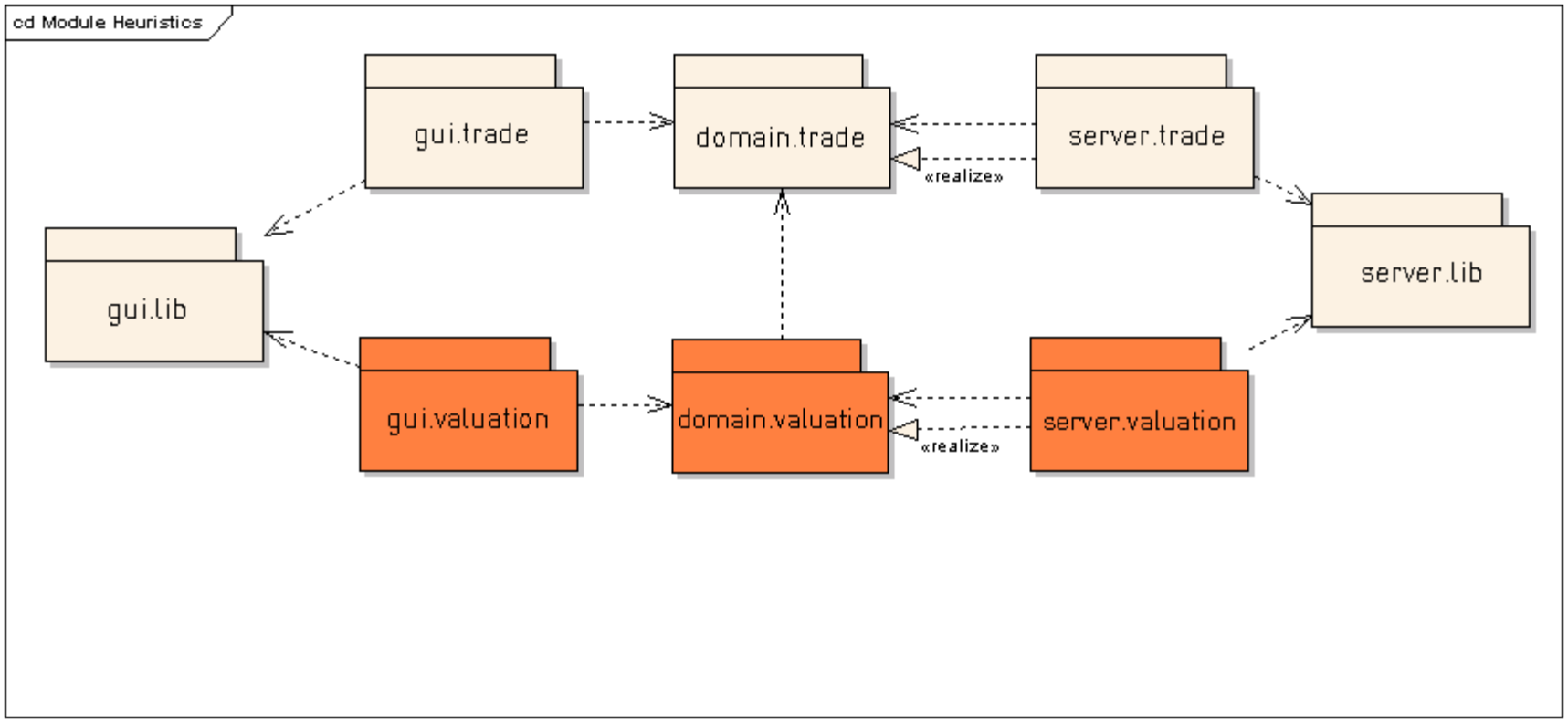
Signs You May Need a New Module

Reuse: functionality required by multiple modules



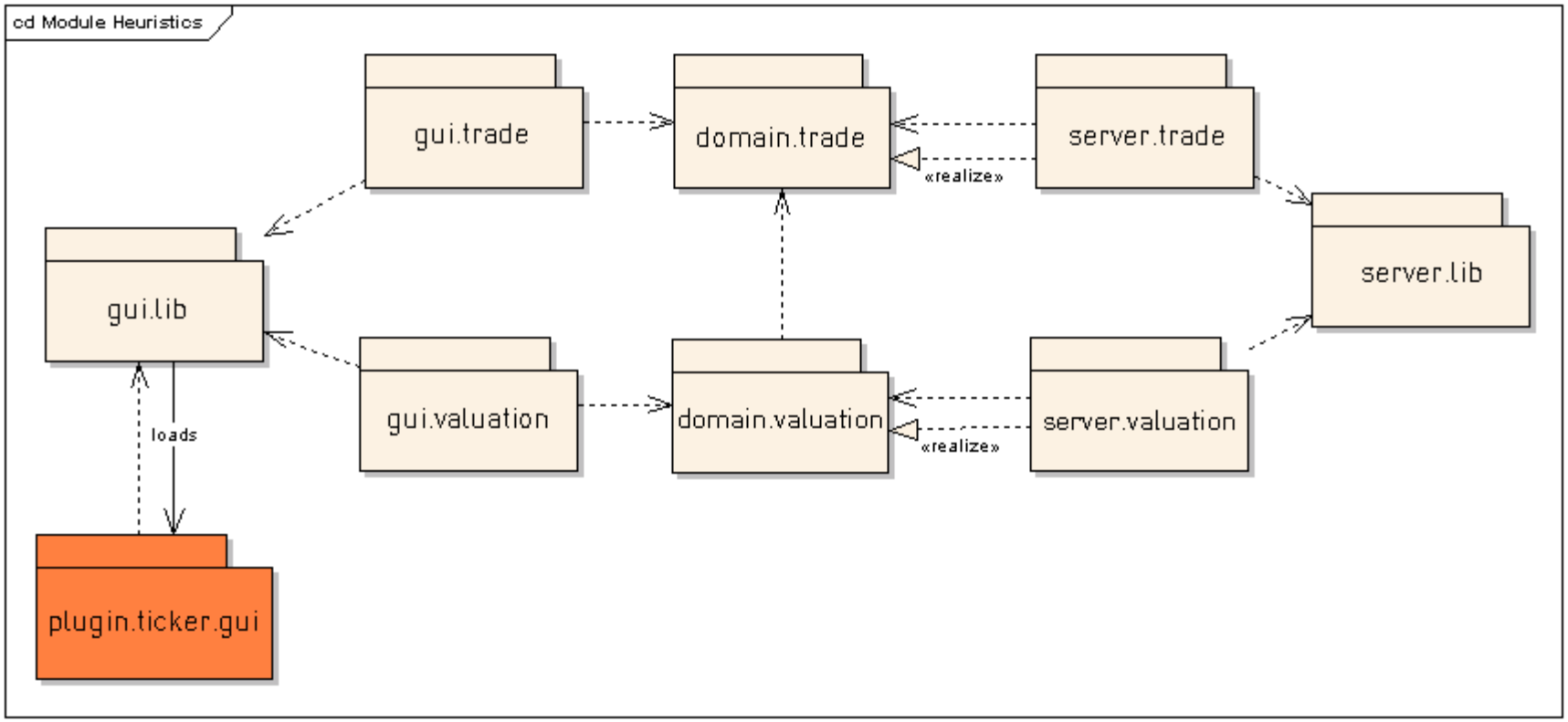
Signs You May Need a New Module

Functional cohesion: different groups of responsibilities



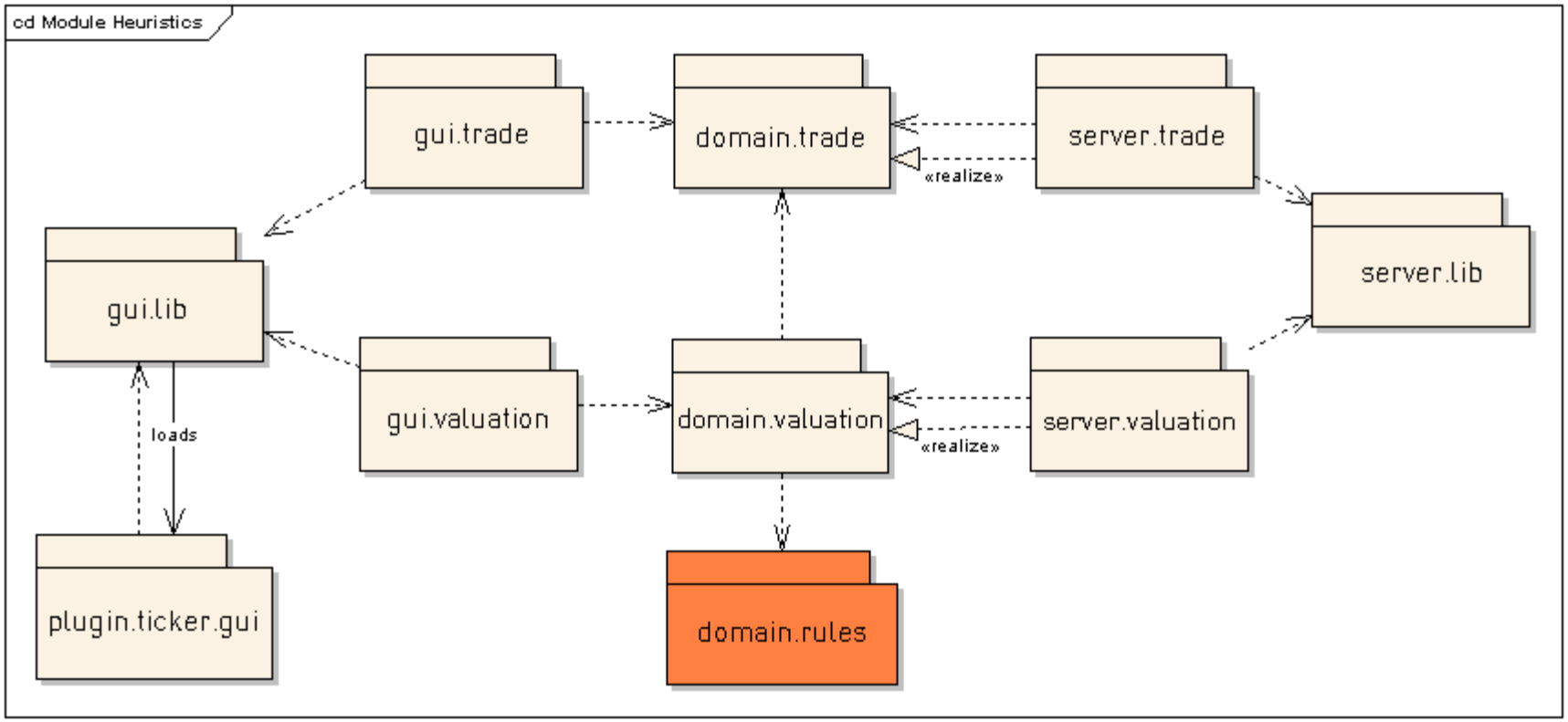
Signs You May Need a New Module

Optionality: functionality not always required or available



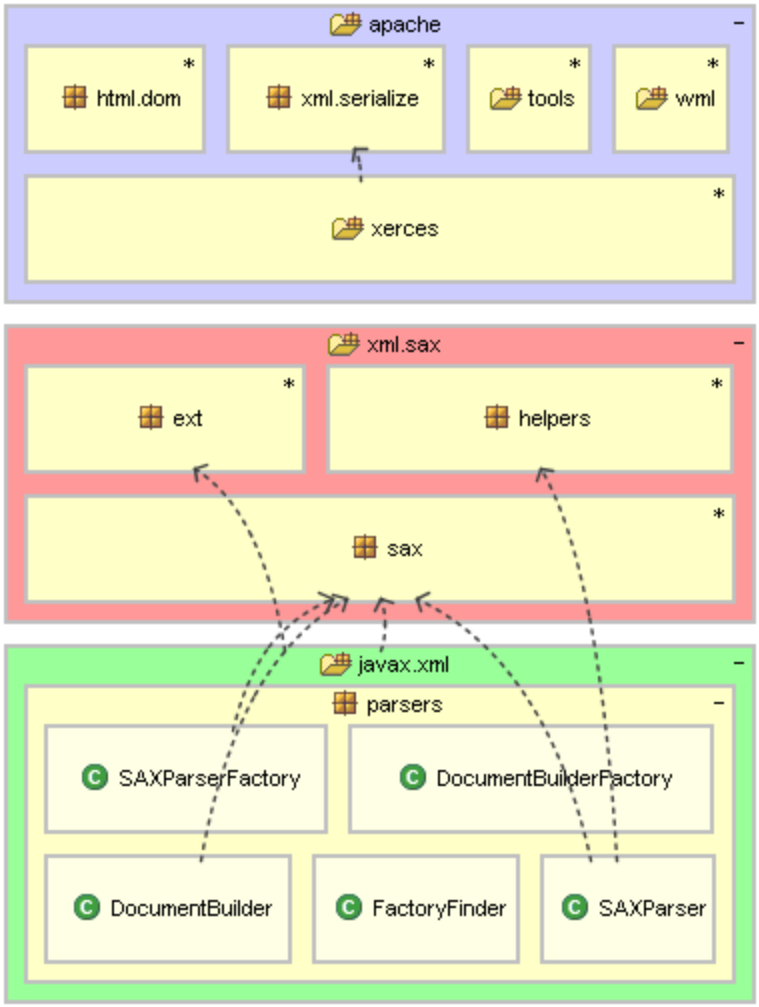
Signs You May Need a New Module

Isolation: protect more stable code from frequent changes



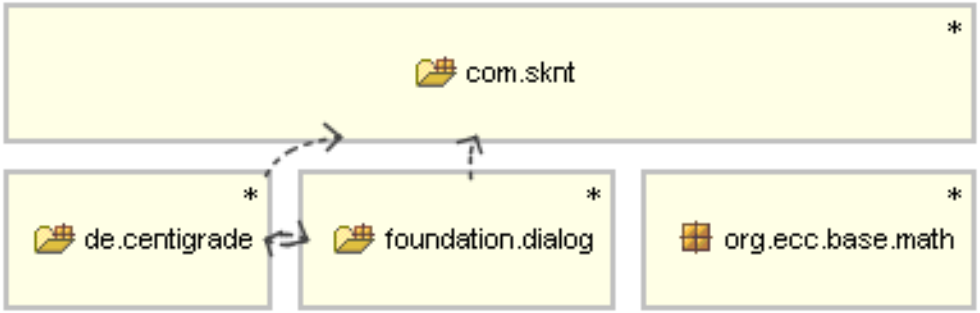
Analysis Methodology – Architecture Diagrams

- Structure101 diagrams define:
  - Compostion
  - Layering
  - Visibility
  - Mapping to code (patterns)
- Show current violations
- Adjustable level of detail
- Spread architecture over multiple diagrams so each is a simple "mind-sized chunk"
- Diagram editor used for analysis + definition



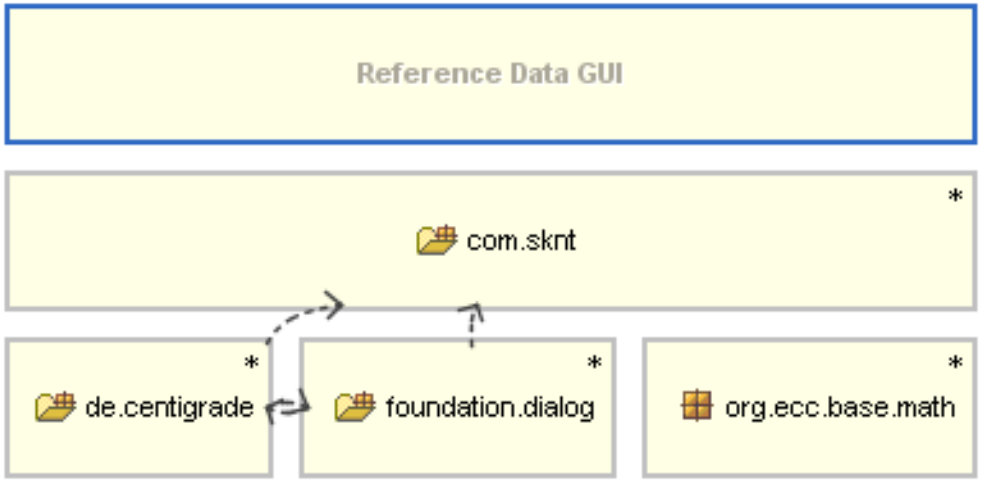
Analysis Methodology

Start with the top-level view provided by Structure101



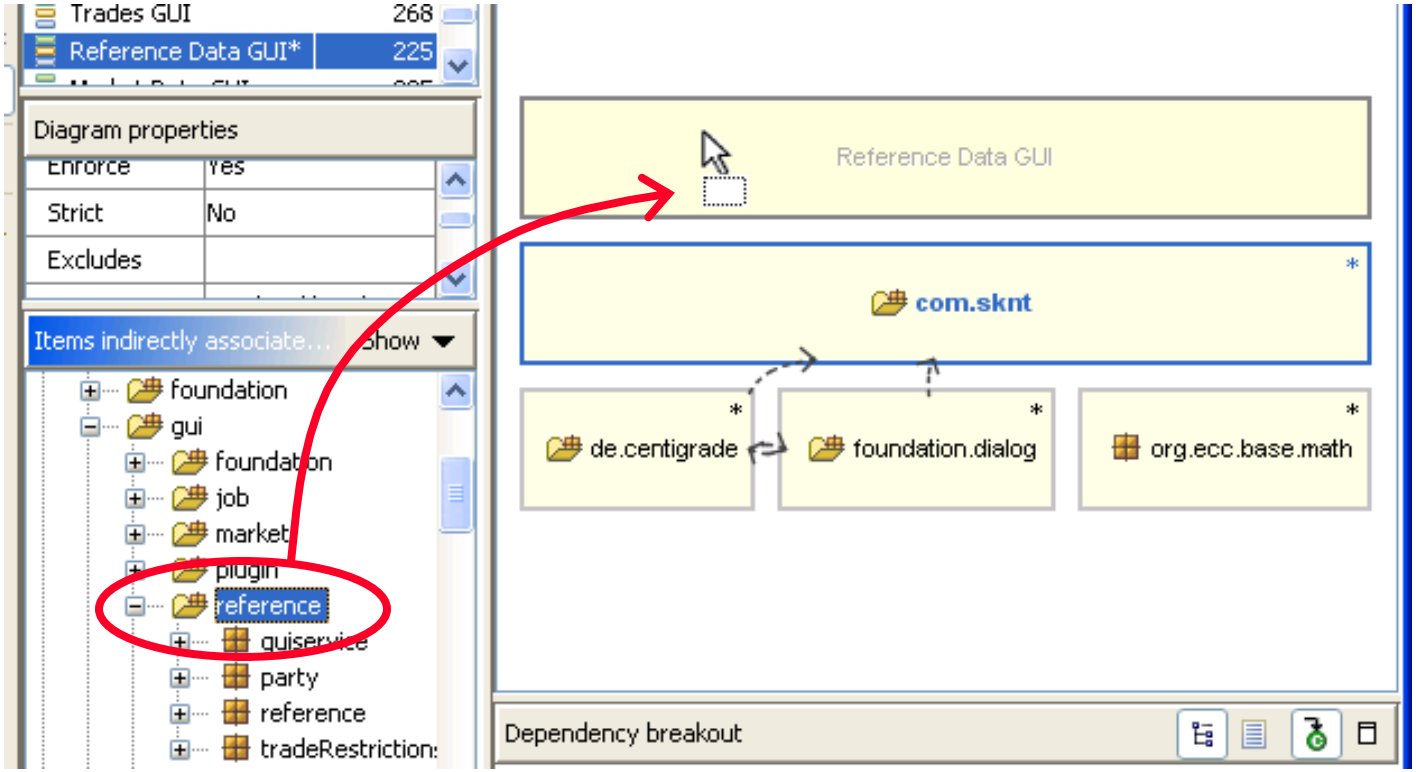
Analysis Methodology

Pick a subject, and add a “bucket” layer for related code



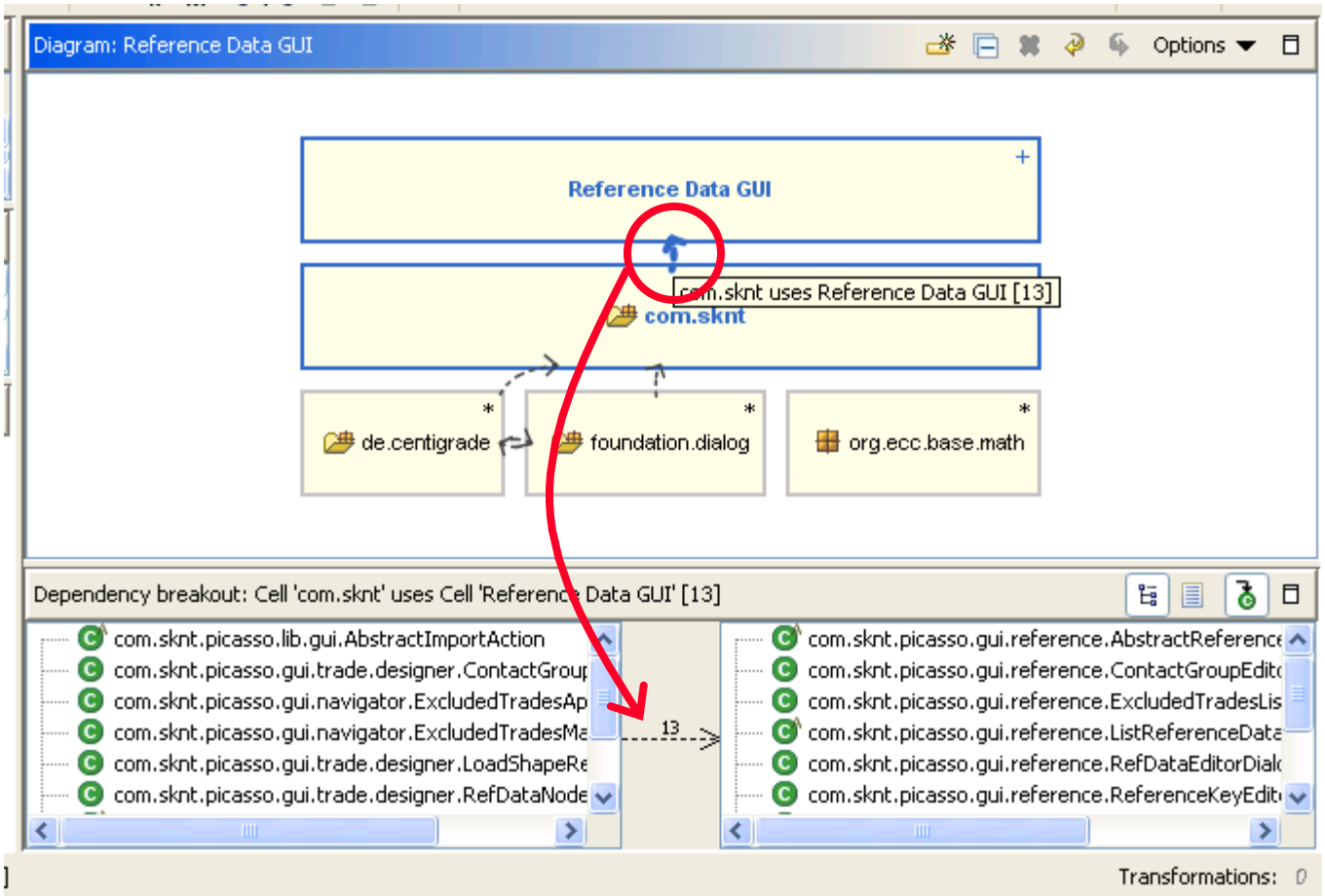
Analysis Methodology

Select related classes and move to the “bucket”



Analysis Methodology

Follow dependencies to find related code

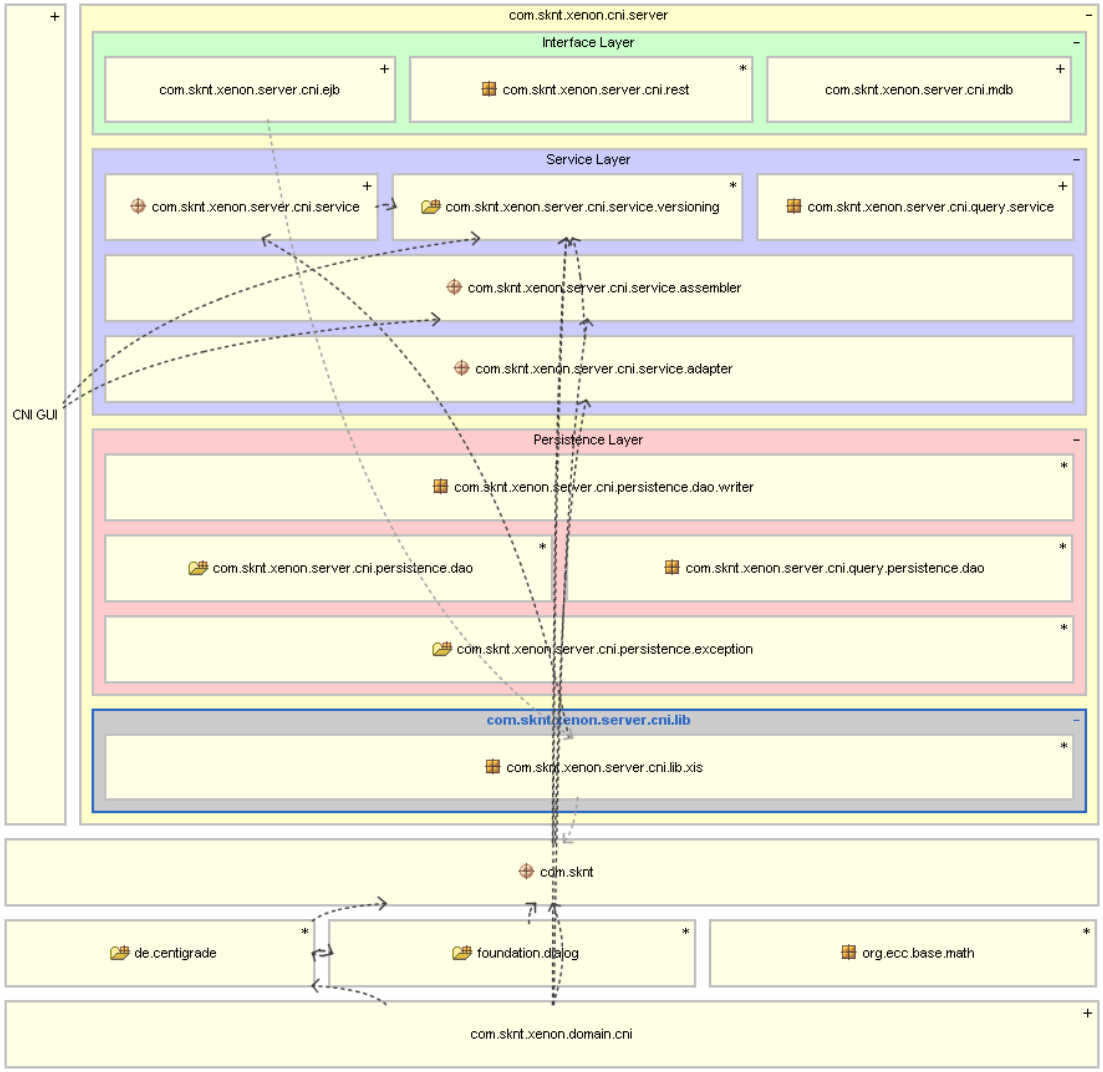


Analysis Methodology

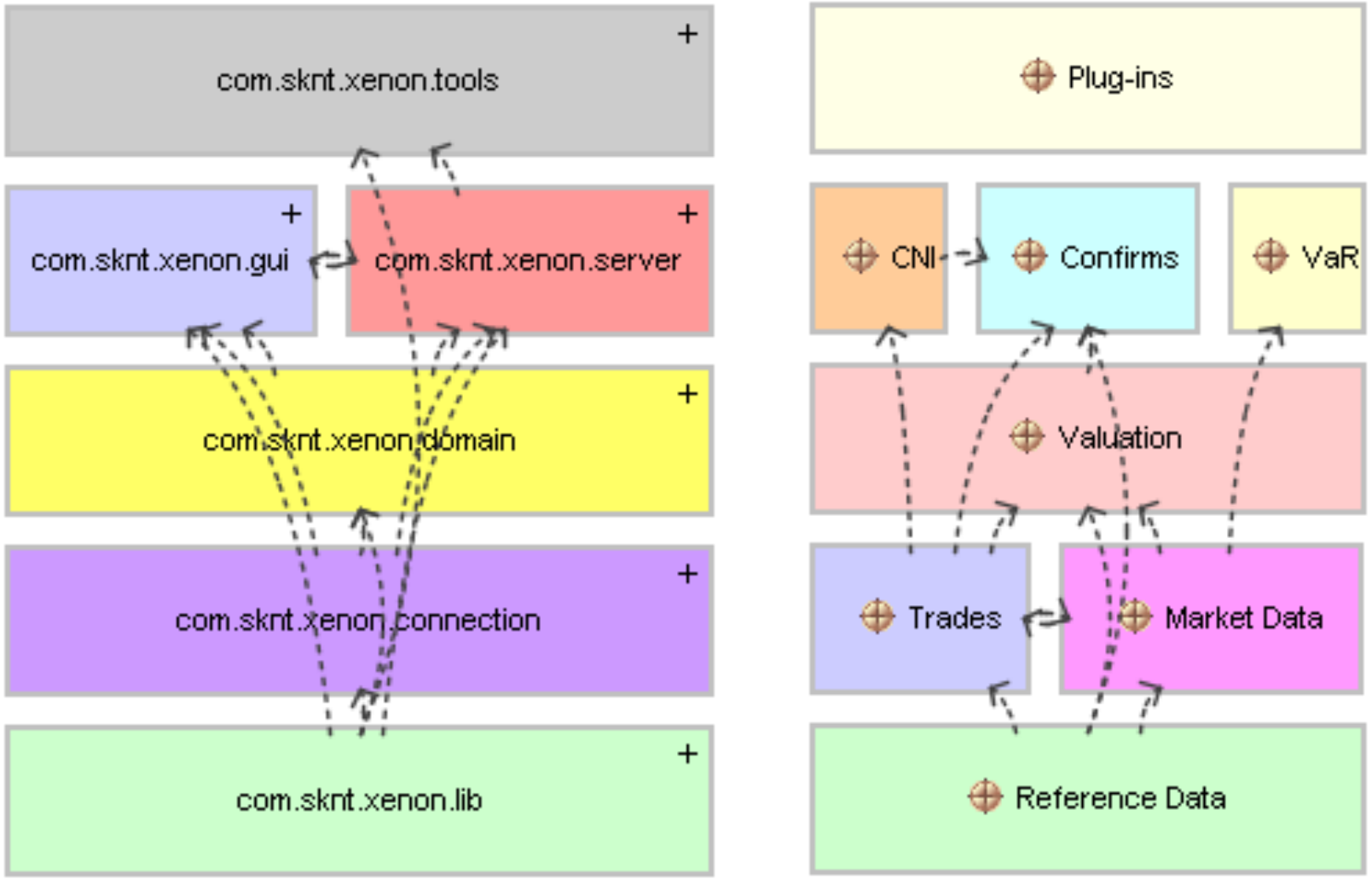
Repeat as necessary until analysis is complete

The resulting diagram serves as a roadmap for refactoring

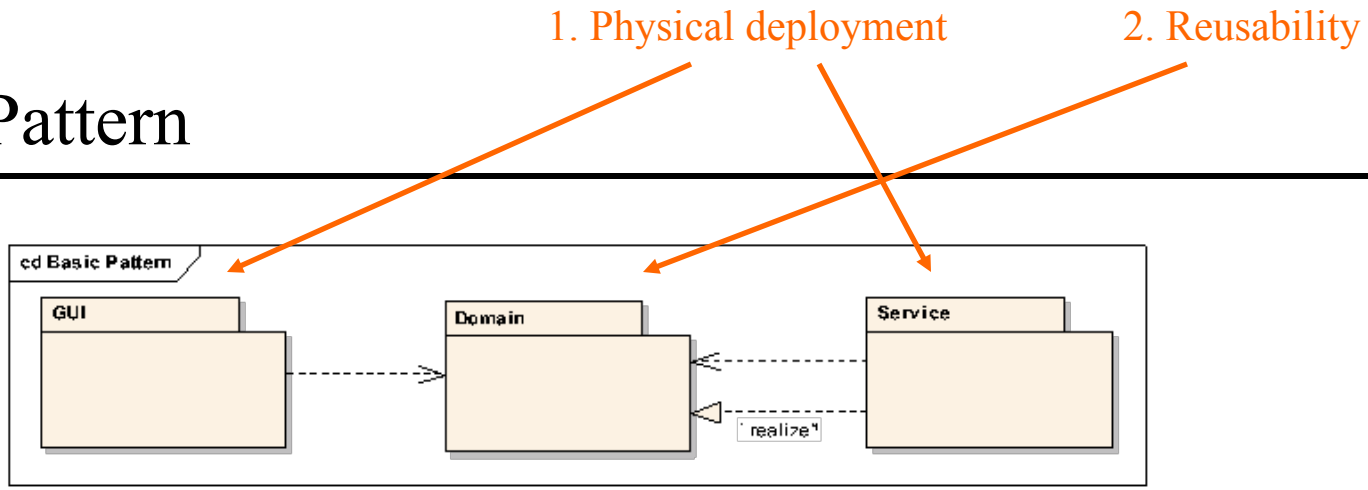
Reverse-dependencies that remain will require modifications to the code



Application Tiers vs. Functional Modules



Basic Pattern



This pattern is repeated for each of the basic functional modules:

– Trades, Market Data, Reference Data, CNI, etc.

GUI

- GUI code
- Uses remote version of service locators
- Depends on gui libraries, domain modules

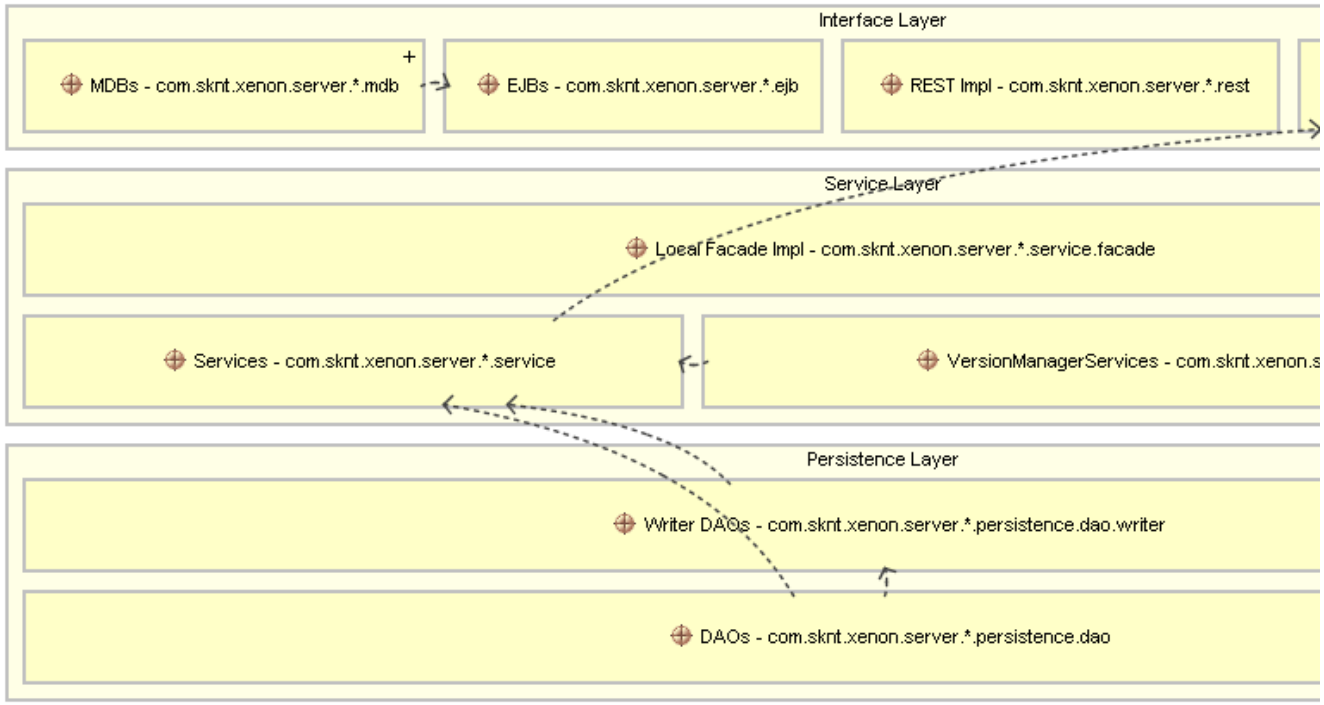
Domain

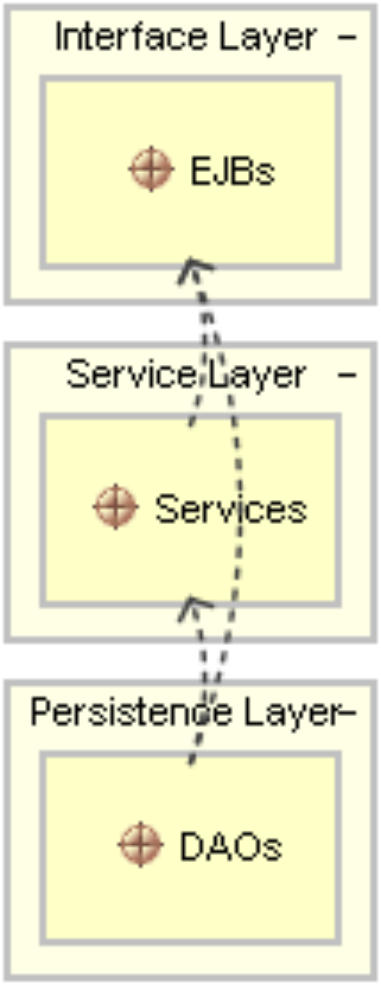
- Business domain objects
- Service interfaces
- Service locator interfaces
- Remote service locator implementation
- May use other domain modules

Server

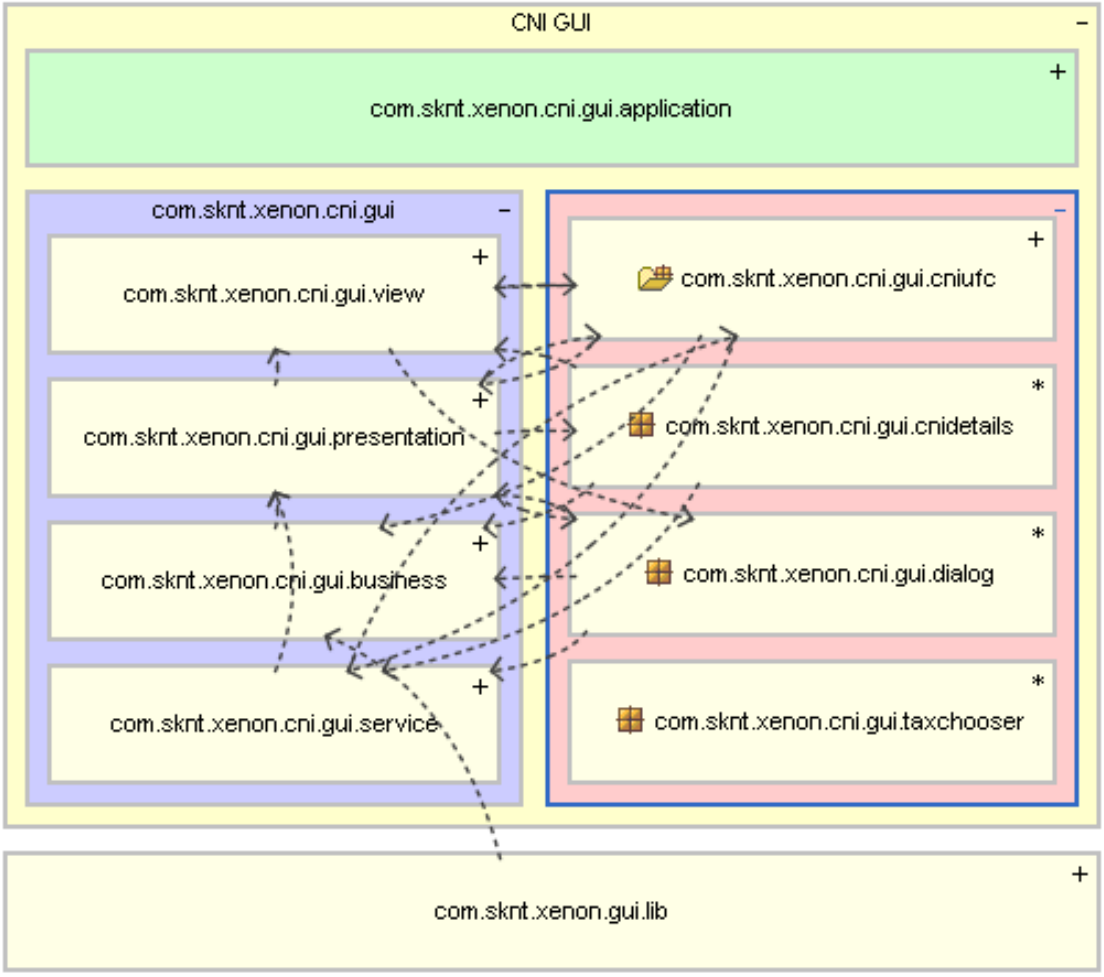
- Service implementations
- May use service interfaces and domain objects of other domain modules
- May use local or remote version of service locators

Server Layers





GUI Layers


CNI GUI
com.sknt.xenon.cni.gui.application
com.sknt.xenon.cni.gui
com.sknt.xenon.cni.gui.view
com.sknt.xenon.cni.gui.presentation
com.sknt.xenon.cni.gui.business
com.sknt.xenon.cni.gui.service
com.sknt.xenon.cni.gui.cniufc
com.sknt.xenon.cni.gui.cnidetails
com.sknt.xenon.cni.gui.dialog
com.sknt.xenon.cni.gui.taxchooser
com.sknt.xenon.gui.lib

Remodularized Packages

`com.sknt.xenon.<module>.<submodule>.<layer/type>`

Examples:

- com.sknt.xenon.domain.trade.lib
- com.sknt.xenon.server.reference.persistence.dao
- com.sknt.xenon.gui.trade.guiservice
- com.sknt.xenon.domain.market.volatility.dto

Benefits:

- Identify module & layer by package name
- Provide better cohesion via smaller packages
- Enable enforcement via simple rules in Structure101
  - E.g. `*.dao.*` is not allowed to use `*.service.*`

Phase II: Preparation & Tools

- Redesign Maven build
  - Maven 2 for more standardization
  - Remove awkward scripts
  - Module = JAR
  - Generate Maven 2 reports
- Hudson
  - Continuous integration
  - Previously, Hudson had to delegate to a perl script
- Structure101
  - IDE Plugins (Eclipse, IntelliJ)
  - Break the build on layer violations
    - Note that only NEW violations should break it!

Phase II: Details

- Work done directly in main branch
  - Old build unaffected
    - POMs don't affect Maven 1 project files
  - Developers hardly noticed
- All "future" modules created in a subdirectory called "modules"
  - Developers not distracted by dozens of empty new modules
  - POMs fully configured, but no code moved
- New root folder created for "packaging"
  - EARs, WARs, etc.
  - Before, was done by custom script
  - Now, Maven 2 standard methods

Phase II: Details

- Rollout plan
  - Present "Remodularization"
    - Where do I put new code?
    - Should developers move code to new modules themselves?
    - How do I find stuff?
    - How do I know what's already been remodularized?
  - Training on new dev environment (changes minor)
  - Update internal documentation
  - Wait for start of next big project before switchover
- First big change:
  - Moved "old" modules to "modules-picasso" subdirectory

Phase III: Remodularization

- Work is still ongoing
- Mini-projects
  - 1-3 developers only
  - Major input from “module owners”
  - Avoid impacting new work, but piggy-back off of other regression testing
- Must be able to stop Remodularization at any milestone and start again later
  - Project can survive shifting priorities
  - Have to prevent decay during downtimes

Phase III: Remodularization

First targets:

- GUI "Tier"
  - Lowest afferent coupling
  - Early benefits (GUI size, code clarity)
  - GUI team more "available"
- CNI Functional Module
  - Architectural "Tracer bullet"
  - Least "legacy" of modules
  - Not conflicting with new development

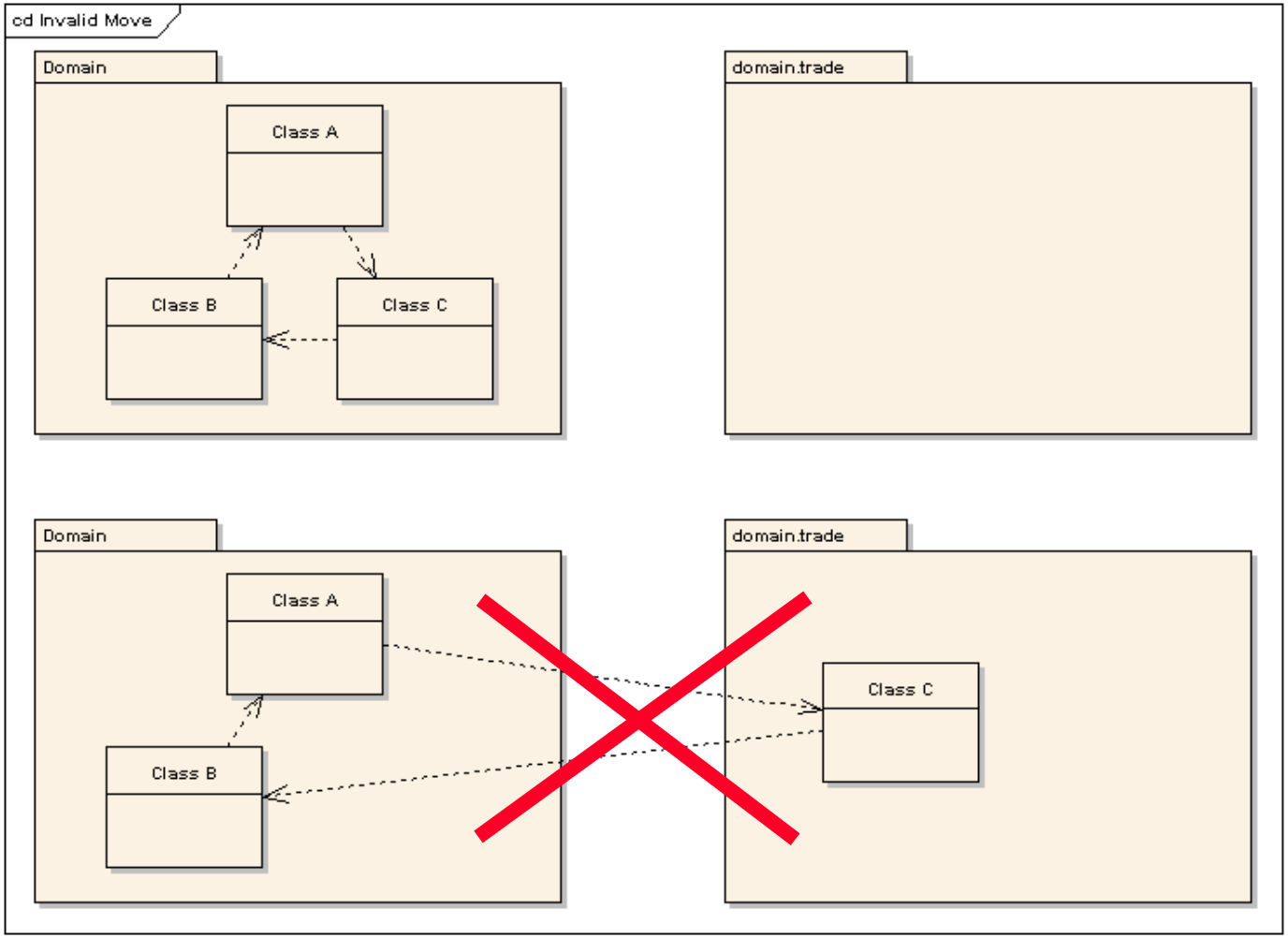
Dealing with Dependency Cycles

Need to avoid creating dependency "tangles" between modules

- A "tangle" is always a violation of the Structure101-defined architecture layering
- A module-level tangle means the application won't compile!
- *Choose:* only old modules depend on new, or new modules depend on old
  - We chose latter approach in order to start at "top" of dependency tree (smaller impact at beginning)
  - Rule of thumb: any class can be moved to the new modules as soon as all the classes which depend on it are in the new modules.
- Tangles need to be moved as a whole, or tangle must be broken

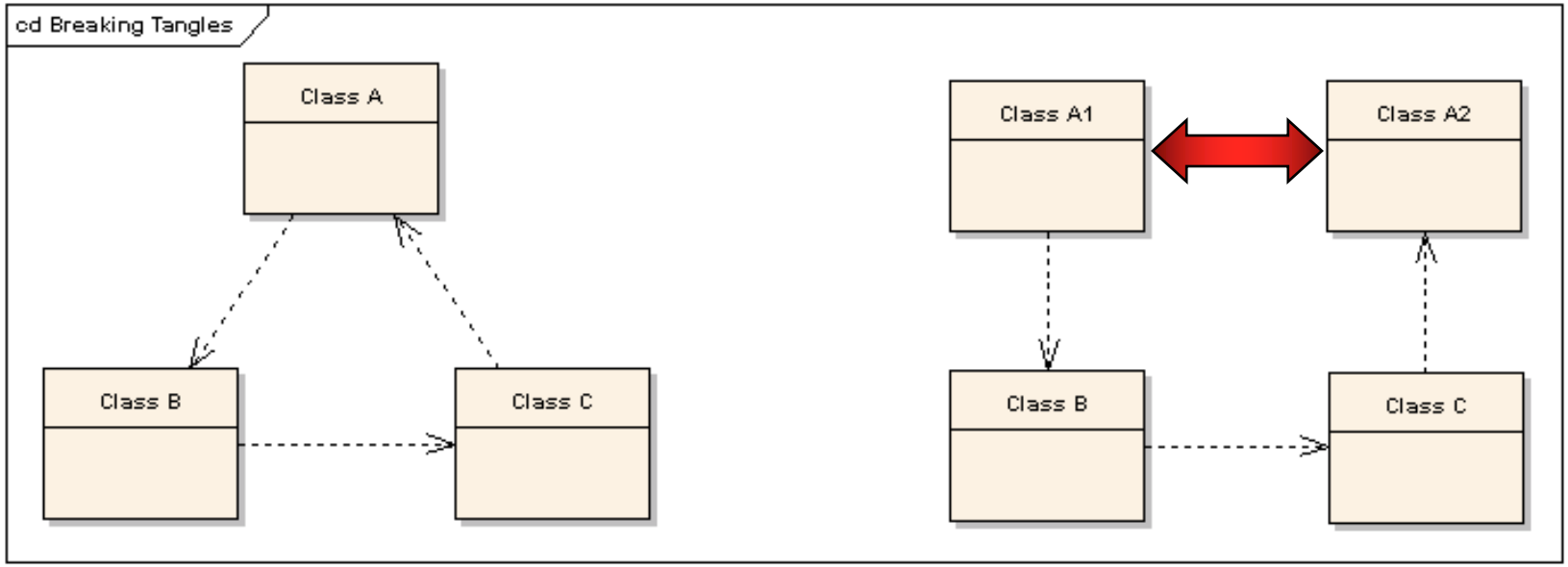
Dealing with Dependency Cycles

Tangles must be moved as a whole, or rewritten



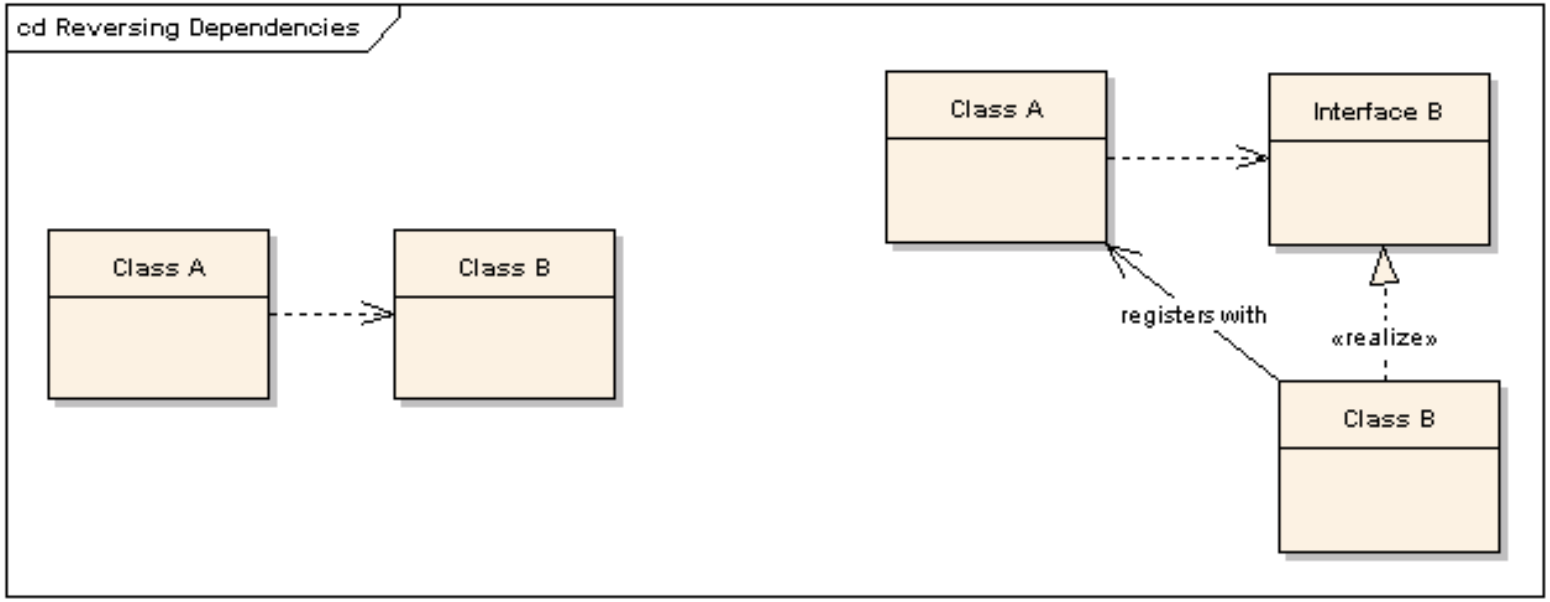
Dealing with Dependency Cycles

Cycles may be broken by splitting a class along lines of responsibility



Dealing with Dependency Cycles

Dependencies may be reversed via observer, registry or factory patterns



Preventing Decay

- Presentations not enough
  - First time this presentation was shown to developers, a test was given. Only about 50% really “got it”.
- Focus on tools for communication & enforcement
  - Structure101 IDE Plug-in
  - Maven build
  - Findbugs
  - SVN commit hooks

Communication & Enforcement

Java - xenon-domain/src/java/com/sknt/picasso/market/dao/HistoricalFixingsDaoComposite.java - Eclipse Platform

File Edit Source Refactor Navigate Search Project Run Window Help

ASPeriodBoundedField | HistoricalFixingsDao

```
            query.append("     AND UNDERLYING.FIXING_AMOUNT
            break;
        case HistoricalFixingsService.ENTERED:
            query.append("     AND UNDERLYING.FIXING_AMOUNT
            break;
```

S101: Dependency on com.sknt.picasso.reference.underlying.HistoricalFixingsService violates architecture.

```
        if (start != null) {
```

Outline | Architecture Dia

Diagram: Service to DAO (5 violations)

Proble | Javado | Declara | Metrics | Tasks | Pollutio | Violatio | Propert | Violatio | Search | Progre

From	To	
com.sknt.picasso.market.dao.HistoricalFixingsDao	com.sknt.picasso.reference.underlying.HistoricalFixingsService	
com.sknt.picasso.market.dao.HistoricalFixingsDaoComposite	com.sknt.picasso.reference.underlying.HistoricalFixingsService	
com.sknt.picasso.query.service.dao.QueryExpressionDaoTestCase	com.sknt.picasso.query.service.UserQueryService	(new)
com.sknt.picasso.trade.liquidation.dao.LiquidationDataWriterDaoHibernate	com.sknt.picasso.market.dataset.MarketDataService	
com.sknt.picasso.trade.liquidation.dao.LiquidationDataWriterDaoHibern...	com.sknt.picasso.market.dataset.MarketDataService	

QA Strategy

- JUnit and FIT tests to enable refactoring with confidence
- Strong QA department
  - Fully-documented suite of regression tests
  - 80% automated
  - Load & performance, integration and stress tests
- Dependency matrix to determine which regression tests required
  - Show Structure101 matrix
  - Needed to maintain additional list of runtime dependencies:
    - Reflection
    - Messaging
    - Configuration files (esp. when class names involved)
    - XML (esp. serialized objects)
- Regression tests often covered by other work

Results

Previously, growth in the code base was accompanied by a growth in complexity

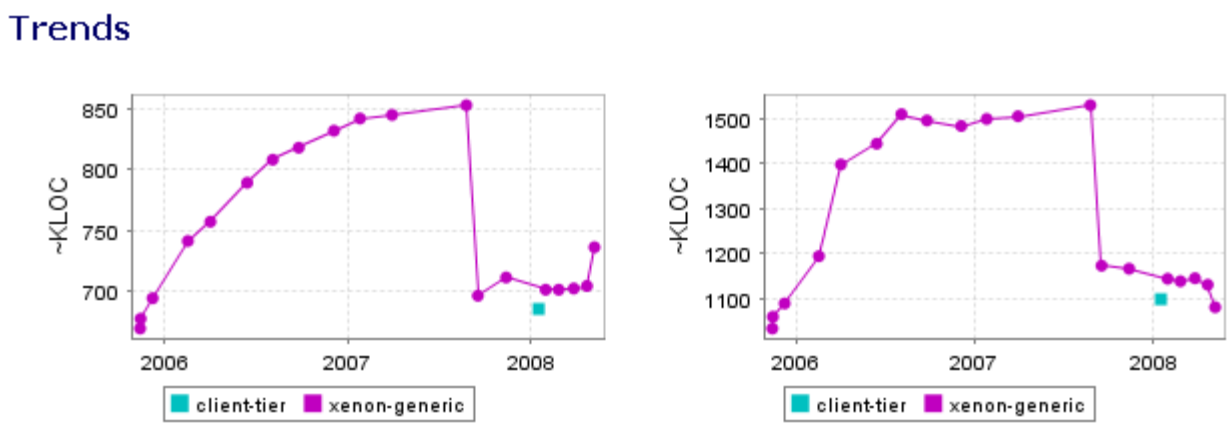


Fig 1: Trends (size)

Fig 2: Trends (XS)

("XS" is a Structure101 measure of structural complexity, including "tangles")

Results

Now, structural complexity is decreasing even as the number of lines increases

Trends

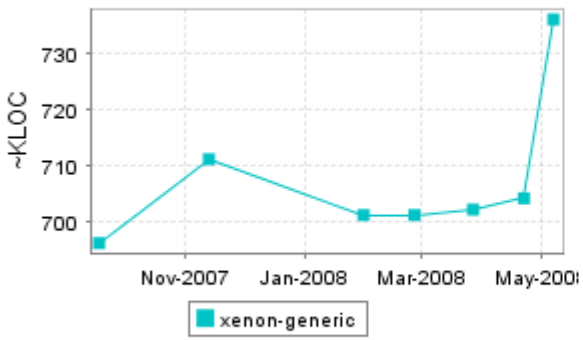


Fig 1: Trends (size)

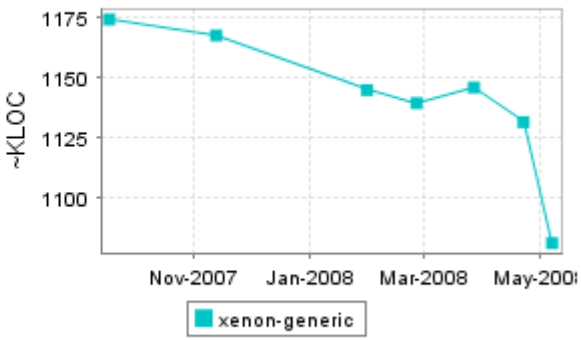


Fig 2: Trends (XS)

Results

Advantages of new model:

- Clearer architecture
  - Can tell module by package name
  - Can tell tier by package name
  - Layering now visible
- Can assign ownership of modules to teams
  - Work allocation view
  - Team leaders can be responsible for maintaining good design, statistics

Results

- Developers now "get it"
  - All on same page
  - Opened the door for new discussions about design
  - Developers know where to put new code
- Reduced dependencies
- Faster build
- Reduced scopes for regression testing
- Decay prevented through instant feedback and enforcement
- Increased application flexibility
  - Standard pattern for new and plug-in modules
  - Many "remodularized" components rewritten with registry features