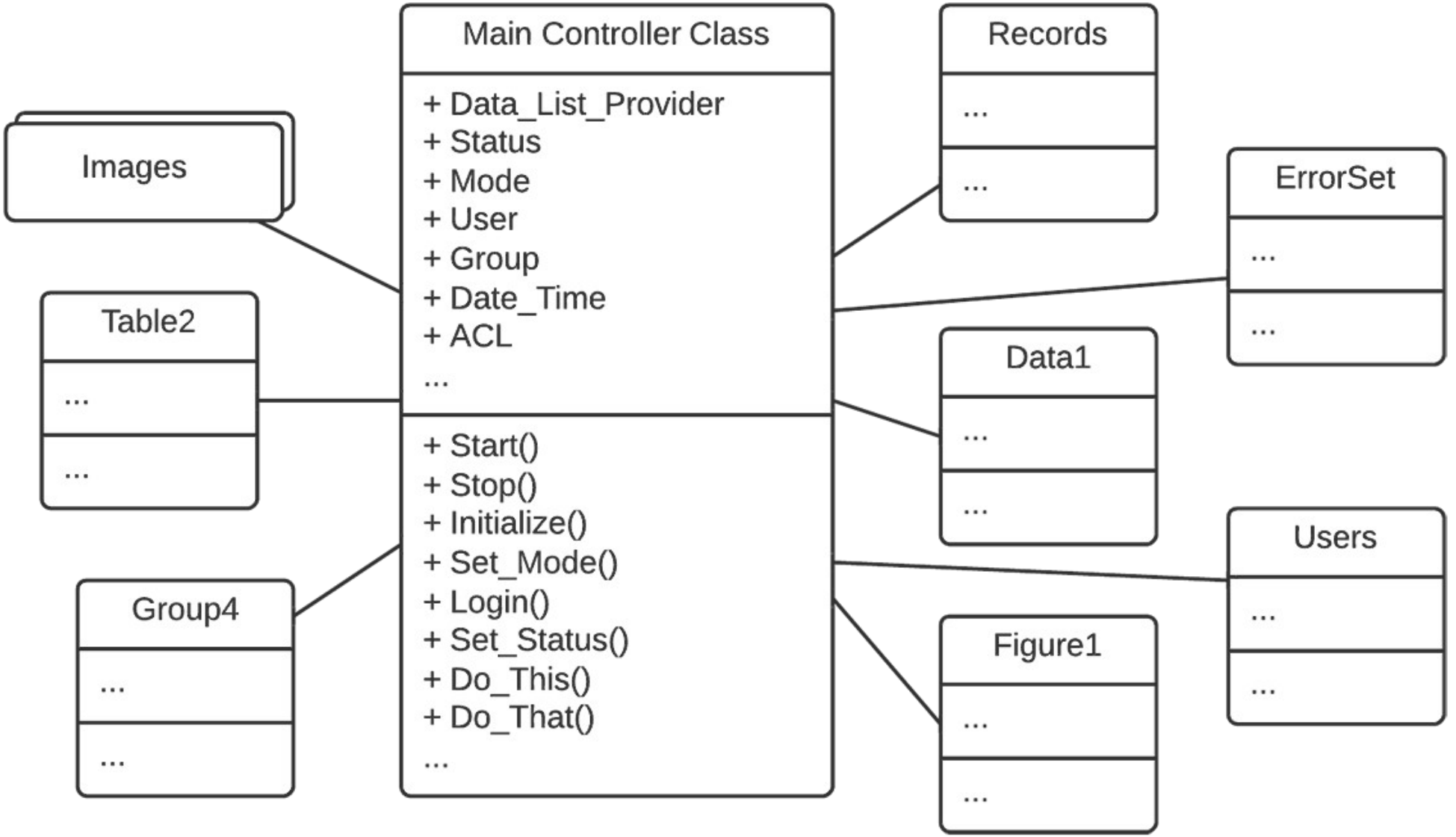
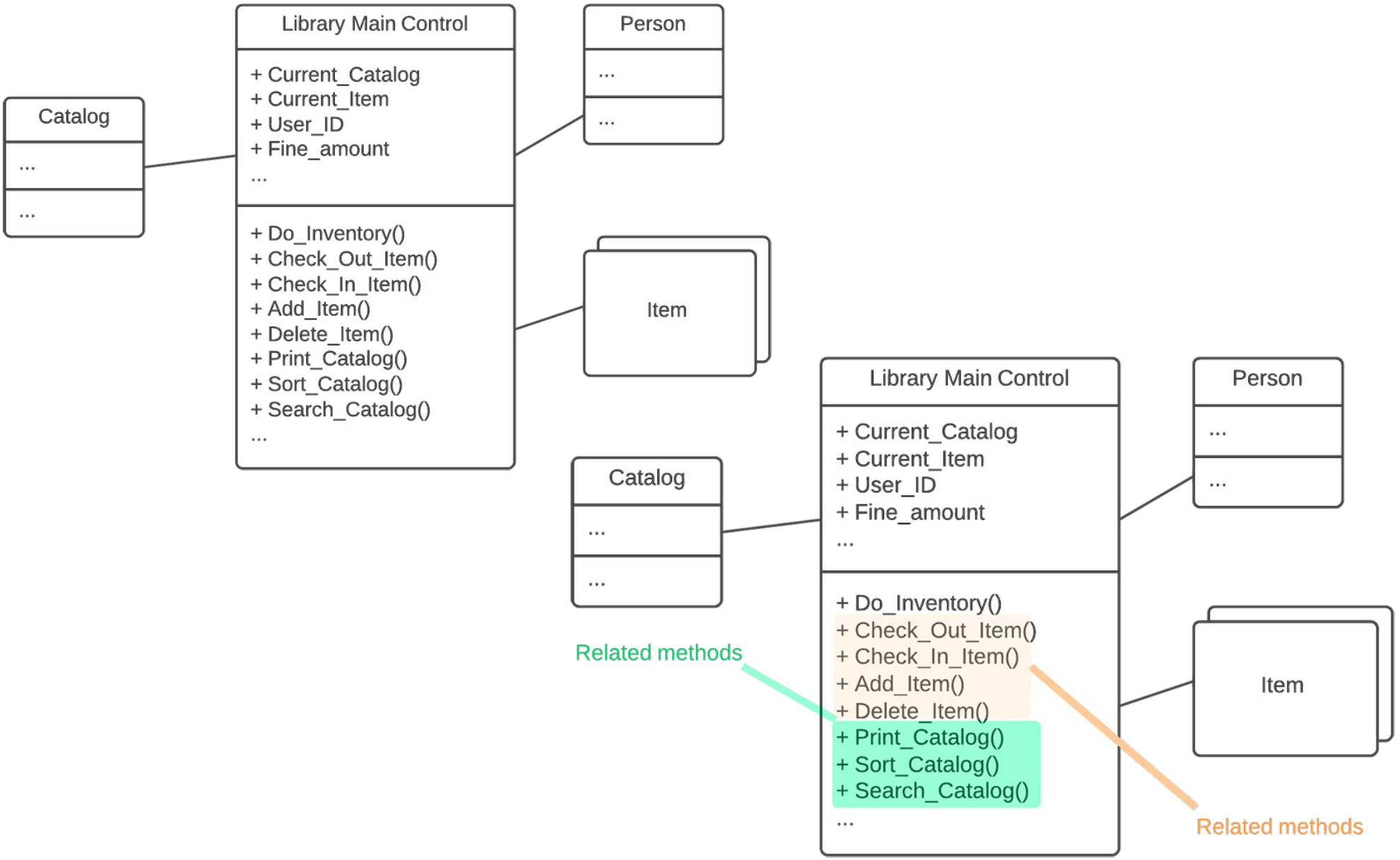
Software Development AntiPatterns: The Blob



Source making https://sourcemaking.com

Software Development AntiPatterns: The Blob



AntiPattern: Continuous Obsolescence

Examples:

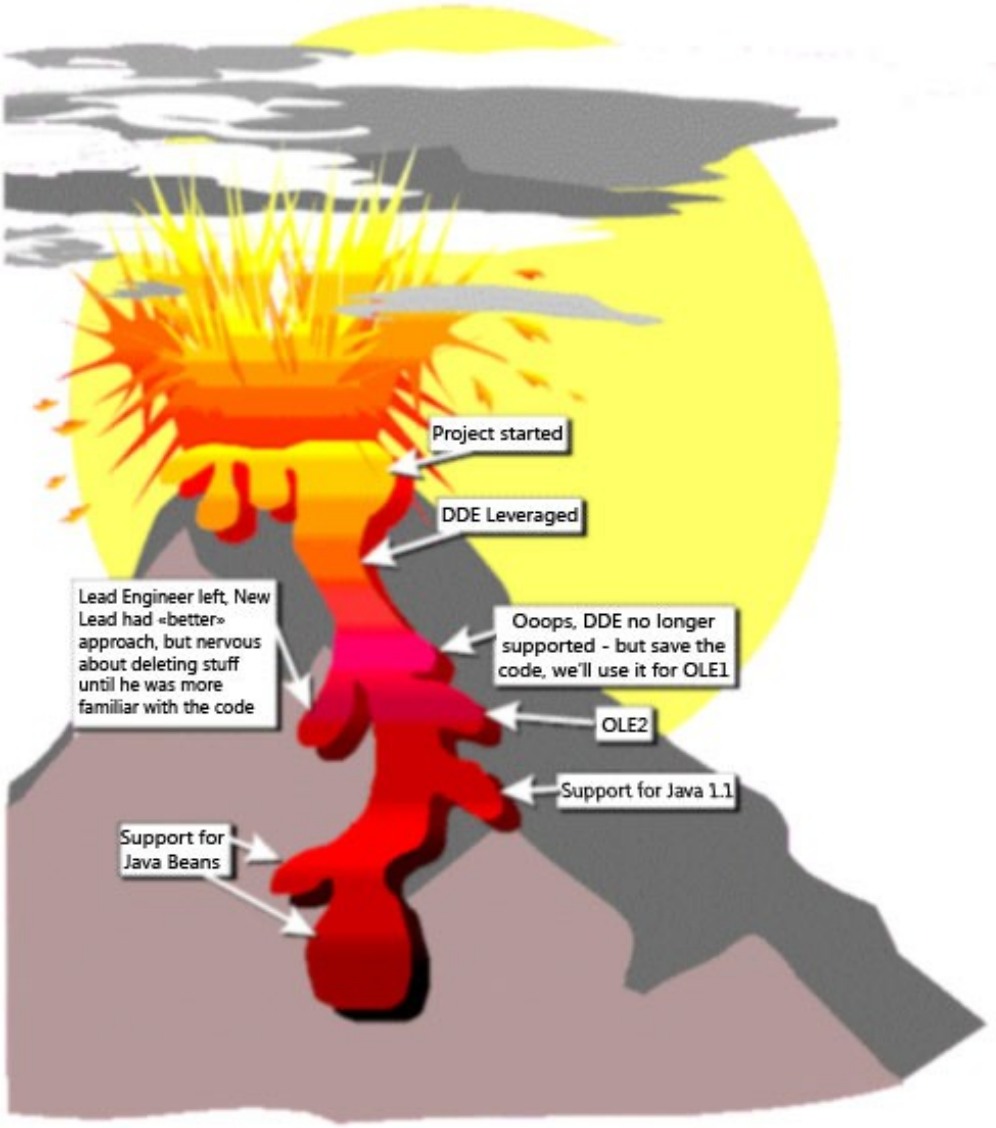
- Java Development Kit — new versions every few months.
- Microsoft dynamic technologies: DDE → OLE 1.0 → OLE 2.0 → COM → ActiveX → DCOM → COM

Refactored Solution

- Use Open Systems Standards.

AntiPattern: Lava Flow

"Oh that! Well Ray and Emil (they're no longer with the company) wrote that routine back when Jim (who left last month) was trying a workaround for Irene's input processing code (she's in another department now, too). I don't think it's used anywhere now, but I'm not really sure. Irene didn't really document it very clearly, so we figured we would just leave well enough alone for now. After all, the bloomin' thing works doesn't it?!"



AntiPattern: Lava Flow

Causes:

- R&D code placed into production.
- Uncontrolled distribution of unfinished code.
- Single-developer (lone wolf) written code.
- Lack of architecture, or non-architecture-driven development.

Refactored Solultion:

- Ensure that sound architecture precedes production code development.

AntiPattern: Ambiguous Viewpoint

Cause:
Object-oriented analysis and design (OOA&D) models are often presented without clarifying the viewpoint represented by the model.

Example:
Models can be less useful if they don't focus on the required perspective(s).
Telephone exchange system perspectives:

- Telephone user.
- Telephone operator.
- Telephone accounting department.

AntiPattern: Functional Decomposition

"This is our ‘main' routine, here in the class called LISTENER."

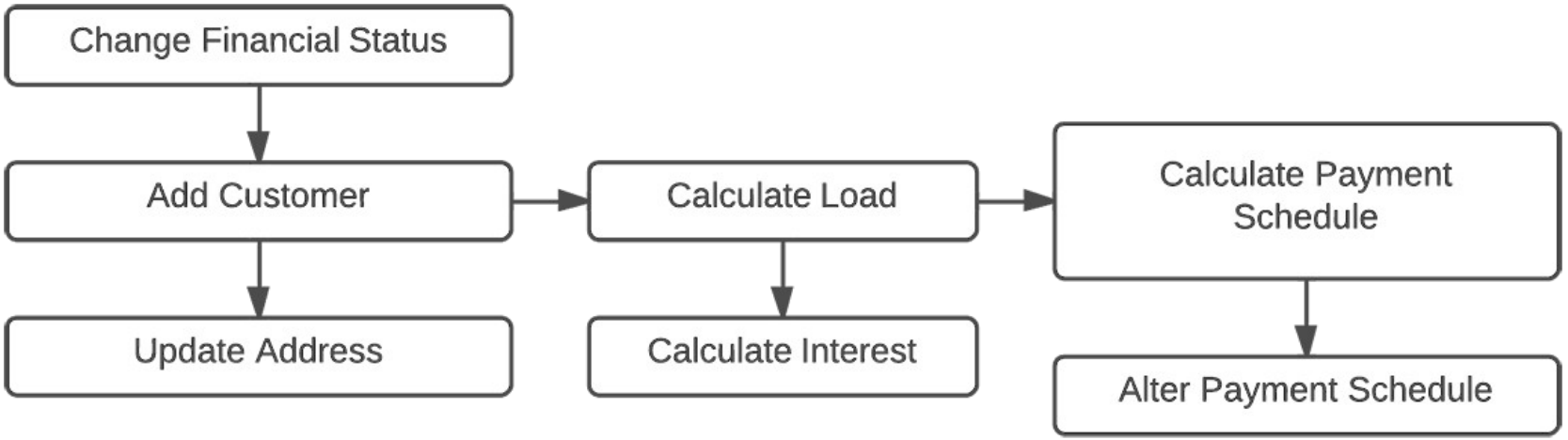
Causes:

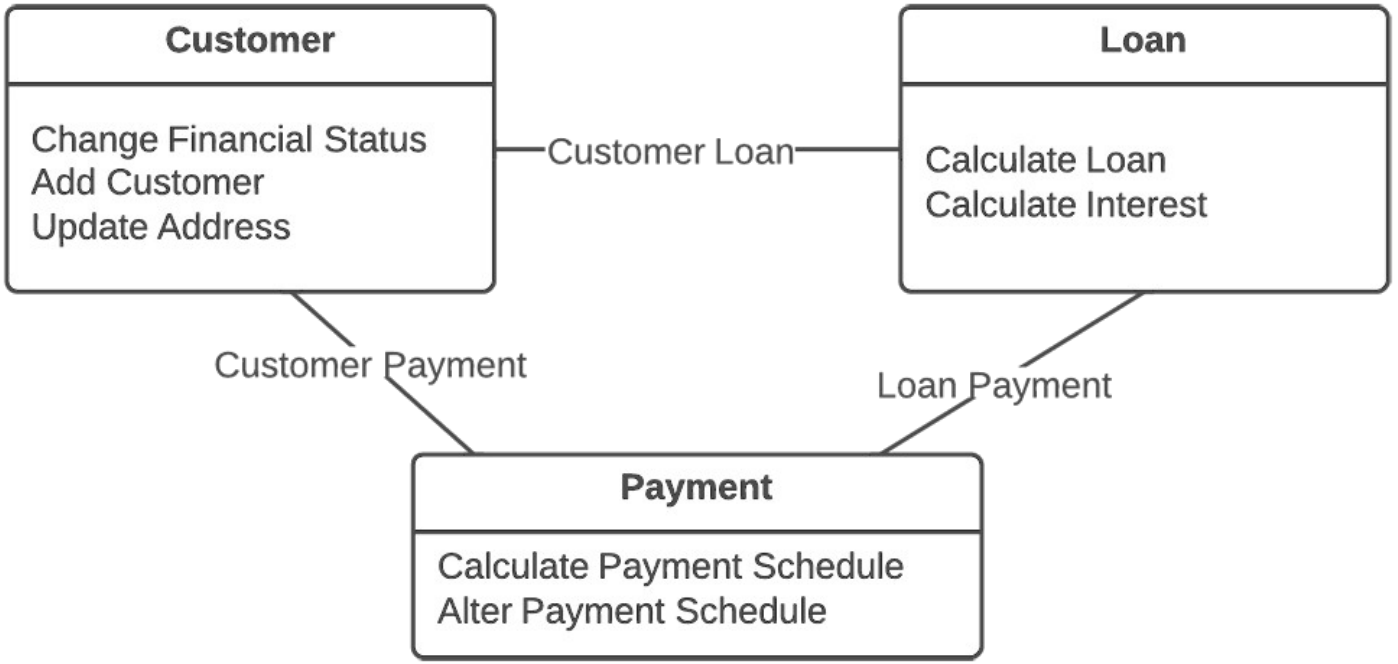
- Lack of object-oriented understanding.
- Lack of architecture enforcement.
- Specified disaster.

Refactored Solultion:

- Discover motivations and provide detailed documentation.
- Formulate design model that incorporates the essential pieces of the existing system.
- Examine the design and find similar subsystems.

AntiPattern: Functional Decomposition





AntiPattern: Poltergeists

"I'm not exactly sure what this class does, but it sure is important!"

Symptoms:

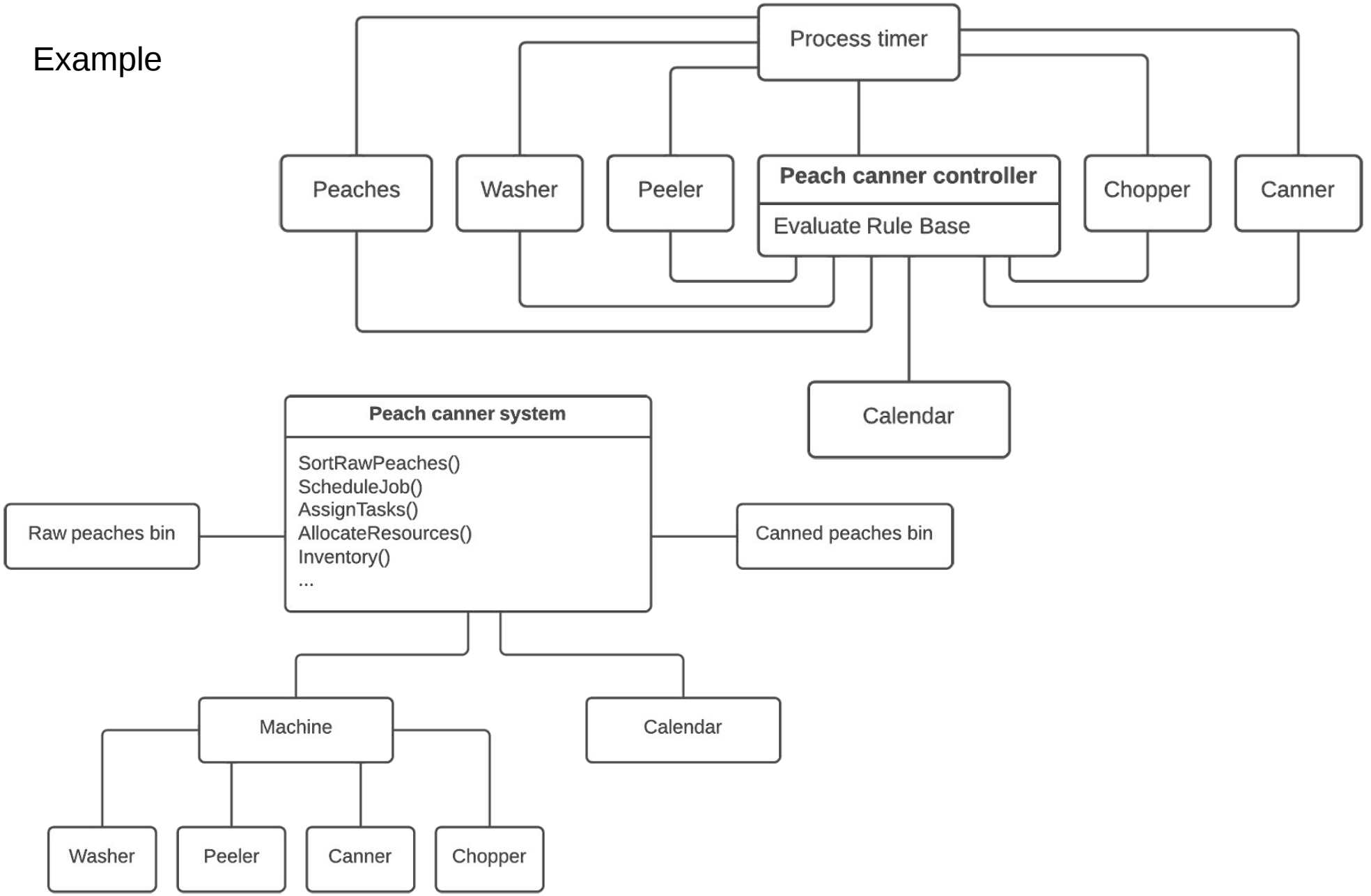
- Redundant navigation paths.
- Transient associations.
- Stateless classes.
- Temporary, short-duration objects and classes.
- Single-operation classes.
- Classes with "control-like" operation names such as start_process_alpha.

Refactored Solultion:

- Removing poltergeist-class
- Replacing provided functionality

AntiPattern: Poltergeists

Example



AntiPattern: Boat Anchor

A Boat Anchor is a piece of software or hardware that **serves no useful purpose on the current project**.
Often, the Boat Anchor is a costly acquisition, which makes the purchase even more ironic.

Refactored Solultion:

- Try to find Boat Anchors prior to acquisition during technology selection process
- Review of product documentation and train-before-you-buy

AntiPattern: Golden Hammer

"I have a hammer and everything else is a nail."
"Our database is our architecture."
"Maybe we shouldn't have used Excel macros for this job after all."

Symptoms:

- Identical tools and products are used for wide array of conceptually diverse products.
- System architecture is best described by a particular product, application suite, or vendor tool set.
- Developers demonstrate a lack of knowledge and experience with alternative approaches.
- Requirements are not fully met, in an attempt to leverage existing investment.
- Existing products dictate design and system architecture.

Refactored Solultion:

- Philosophical aspect as well as a change in the development process

AntiPattern: Dead End

A Dead End is reached by modifying a reusable component, if the modified component is no longer maintained and supported by the supplier.

Refactored Solultion:

- Minimize the risk of a Dead End by using mainstream platforms and Commercial off-the-shelf (COTS) infrastructure

AntiPattern: Spaghetti Code

"Ugh! What a mess!" "You do realize that the language supports more than one function, right?" "It's easier to rewrite this code than to attempt to modify it." "Software engineers don't write spaghetti code."

Spaghetti Code appears as a program or system that contains very little software structure.

Refactored Solultion:

- Software refactoring (or code cleanup).

AntiPattern: Input Kludge

"End users can break new programs within moments of touching the keyboard."

Refactored Solultion:

- For nondemonstration software, use production-quality input algorithms such as *lex* and *yacc*.

AntiPattern: Walking through a Minefield

Numerous bugs occur in released software products; in fact, experts estimate that original source code contains two to five bugs per line of code.

Refactored Solultion:

- Test automation, test design

AntiPattern: Cut-And-Paste Programming

"Hey, I thought you fixed that bug already, so why is it doing this again?"
"Man, you guys work fast. Over 400,000 lines of code in three weeks is outstanding progress!"

It comes from the notion that it's easier to modify existing software than program from scratch.

Refactored Solultion:

- In white-box reuse, developers extend systems primarily though inheritance.
- Refactor the code base into reusable libraries or components that focus on black-box reuse of functionality

AntiPattern: Mushroom Management

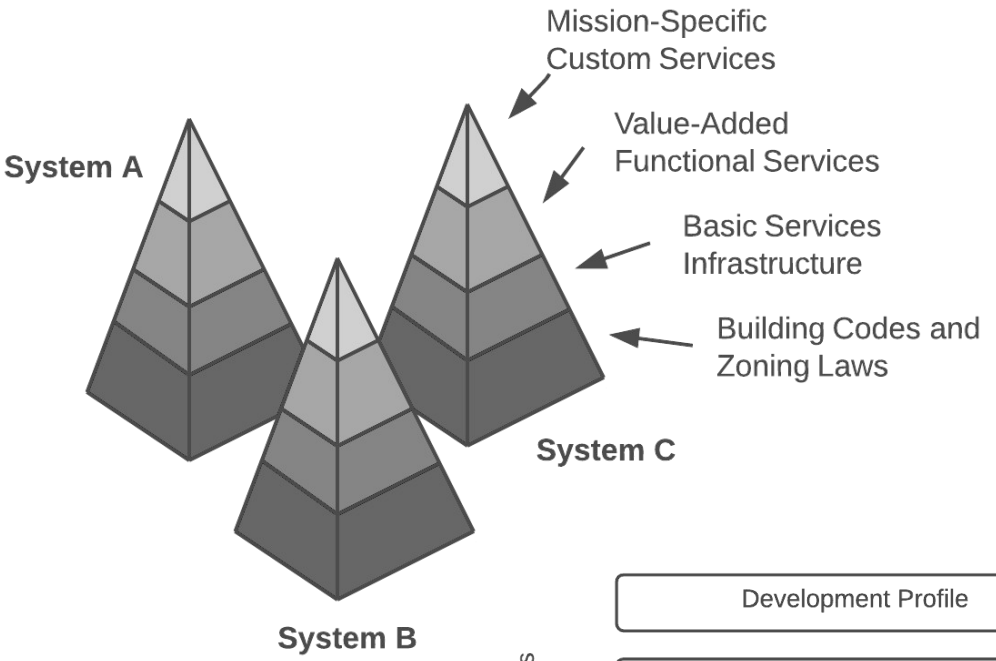
"Keep your developers in the dark and feed them fertilizer." "Never let software developers talk to end users." Furthermore, without end-user participation, "The risk is that you end up building the wrong system."

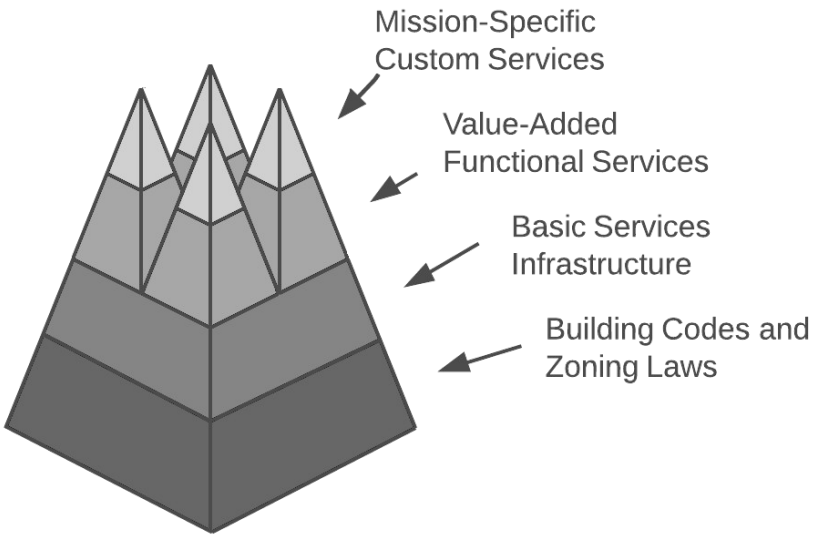
- In reality, requirements change frequently and drive about 30 percent of development cost.
- When developers don't understand the overall requirements of a product, they rarely understand the required component interaction and necessary interfaces.
- Better for user to provide him prototype than documentation.

Refactored Solultion:

- Risk-driven development is most applicable to applications, which are user-interface-intensive and require relatively simple infrastructure support.

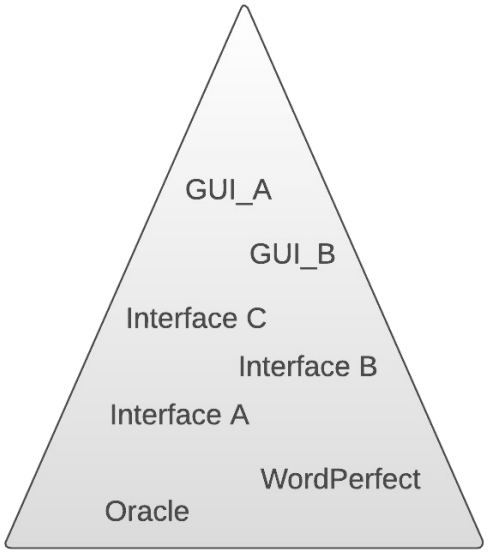
AntiPattern: Stovepipe Enterprise



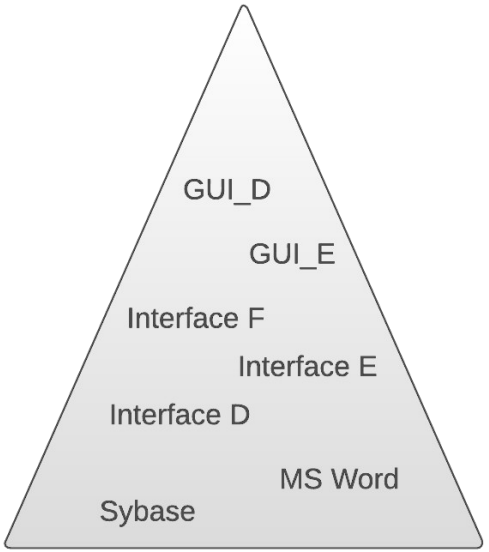


Specifications	Development Profile	1 per system family
	Interoperability Specifications	1 per key point of interoperability
	Computational Facilities Architecture	1 per system family
	Enterprise Architectures	1 per enterprise, family, and system
Requirements	System Requirements Profile	1 per system family
	Operating Environment	1 per enterprise
	Technology Profile	1 per enterprise
	Open Systems Reference Model	1 per enterprise

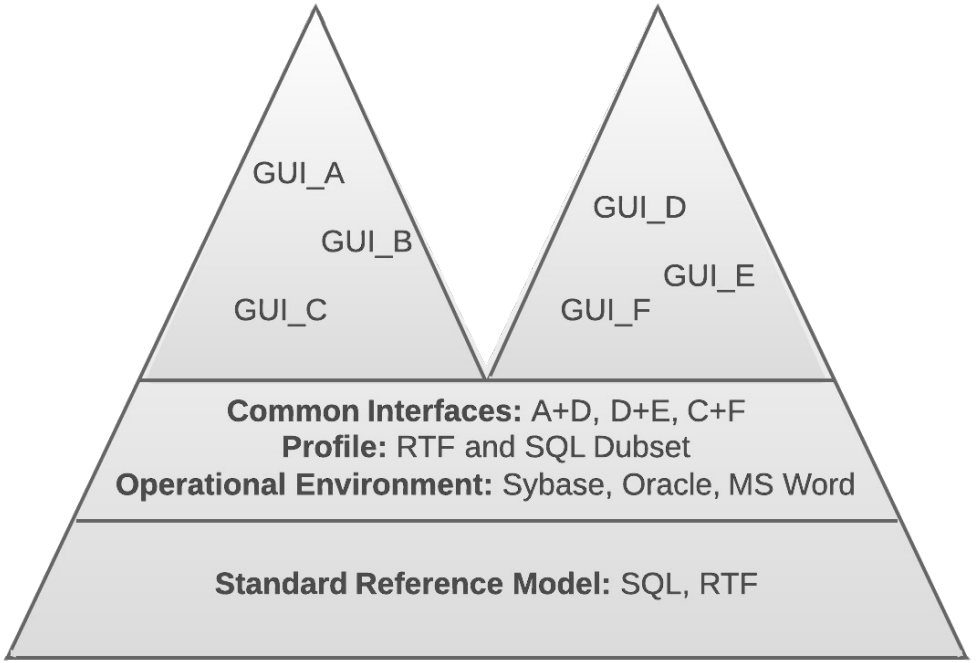
AntiPattern: Stovepipe Enterprise



System 1



System 2



AntiPattern: Stovepipe System

"The software project is way over-budget; it has slipped its schedule repeatedly; my users still don't get the expected features; and I can't modify the system. Every component is a stovepipe."

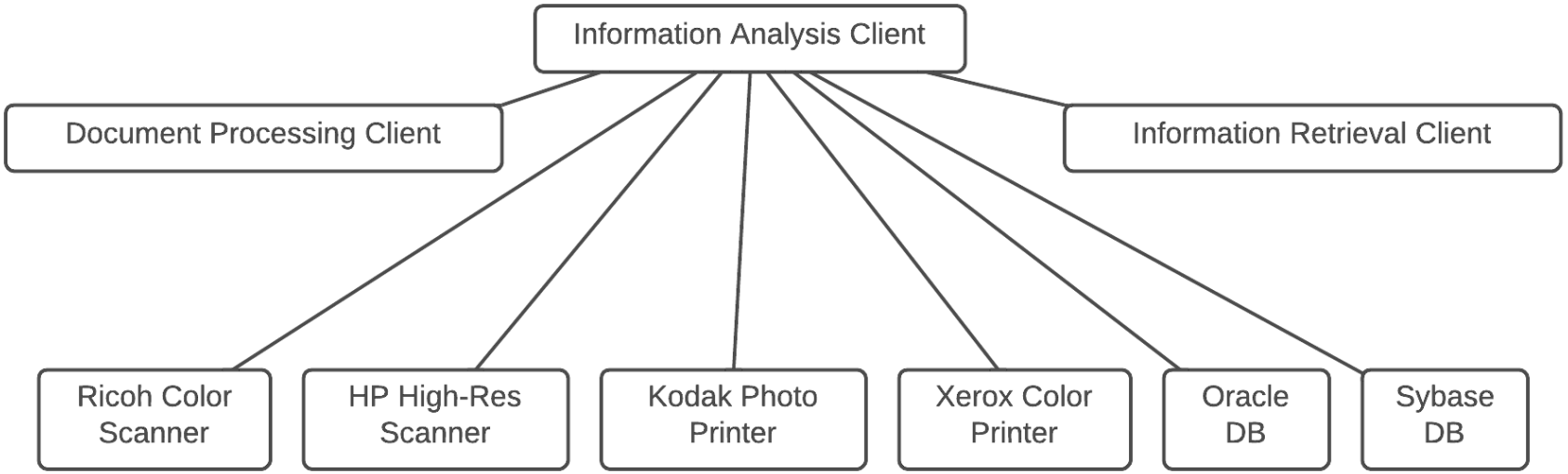
Causes:

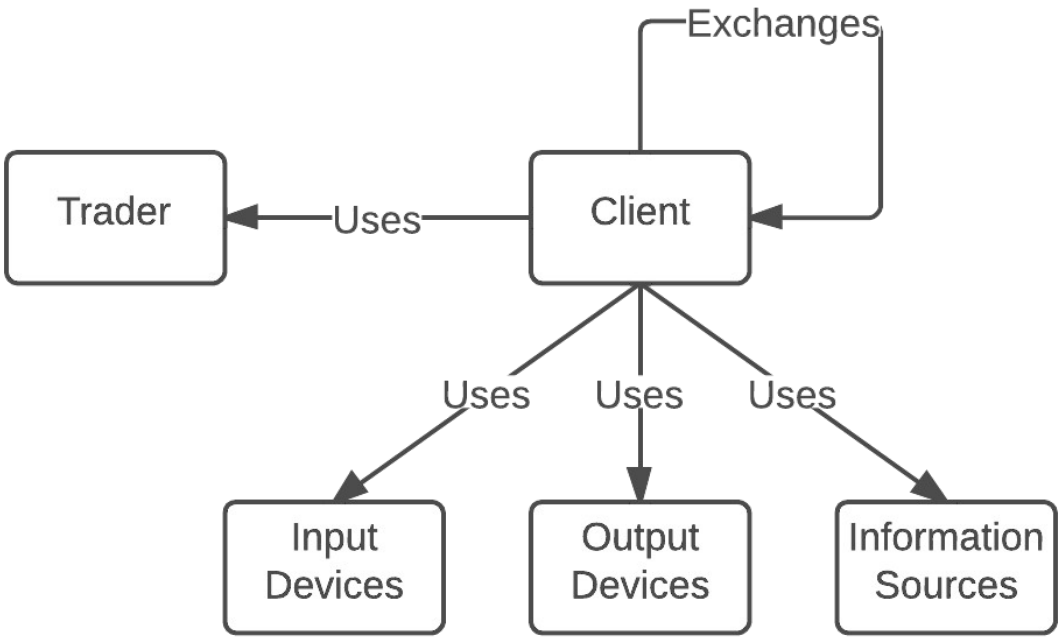
- Multiple infrastructure mechanisms used to integrate subsystems; absence of a common mechanism makes the architecture difficult to describe and modify.
- Lack of abstraction; all interfaces are unique to each subsystem.
- Insufficient use of metadata; metadata is not available to support system extensions and reconfigurations without software changes.
- Tight coupling between implemented classes requires excessive client code that is service-specific.
- Lack of architectural vision.

Refactored Solultion:

- Component architecture that provides for flexible substitution of software modules.

AntiPattern: Stovepipe System





AntiPattern: Cover Your Assets

When no decisions are made and no priorities are established, the documents have limited value. It is unreasonable to have hundreds of pages of requirements that are all equally important or mandatory.

Refactored Solultion:

- Small set of diagrams and tables that communicate the operational, technical, and systems architecture of current and future information systems

AntiPattern: Vendor Lock-In

We have often encountered software projects that claim their architecture is based upon a particular vendor or product line.

Causes:

- The product is selected based entirely upon marketing and sales information, and not upon more detailed technical inspection.
- There is no technical approach for isolating application software from direct dependency upon the product.
- Application programming requires in-depth product knowledge.
- The complexity and generality of the product technology greatly exceeds that of the application needs

Refactored Solultion:

- Isolation of application software from lower-level infrastructure.
- There a need for consistent handling of the infrastructure across many systems.
- Multiple infrastructures must be supported

AntiPattern: Vendor Lock-In