There is nothing permanent except change.

[Heraclitus, 535–475 BC]
Motivation

After a while, many architectures tend to look like this one ...

- The original architecture vision is hardly visible.
- Design flaws are scaffold by many small and local "corrections."
- Missing parts are attached via backpacks.

However:

Such an architecture is doomed to fail before it goes into implementation or operation, because it suffers from:

- developmental qualities like flexibility and maintainability.
- operational qualities like performance and scalability.
Refinements and Refactorings

Therefore:
Create a software architecture step-wise via a number of well-defined, small increments. Each increment includes:
- top-down refinement activities to detail and complete the software architecture.
- bottom-up refactoring activities to garden and clean-up inconsistent or insufficient design decisions.

The process stops if the software architecture is complete and consistent in all its parts and details.

Types of Refactoring

- Refactoring denotes the semantic-invariant transformation of a software artifact
- Most people have read Martin Fowler’s excellent book on Code Refactoring
- Some engineers do also know Joshua Kerievsky’s book on refactoring to patterns
- However, refactoring is not constrained to code. Refactoring may also address
t  - design artifacts such as UML diagrams, models, aspects, DSL expressions, rules, policies. This is Software Architecture Refactoring
  - documents
  - processes
  - test plans
  - databases (⇒ Scott Ambler)
- Unfortunately, current literature is mostly focusing on code refactoring
Documenting Refactorings

- Refactorings can be considered Transformation Patterns
- Thus, it is helpful to provide a canonical form of documentation with sections such as
  - Context
  - Problem
  - General Solution
  - Example
  - „Implementation“ Hints
  - Variants
  - Related Refactorings
- Refactorings may be hierarchical, e.g. for providing refactoring A you may use refactorings B and C

Finding Refactorings

- Because refactorings resemble patterns, they are not invented but found
- Hence, what Brian Foote once said about patterns also holds for refactorings „... are an aggressive disregard of originality“
- Sources to obtain patterns are collections of best practices such as
  - Programming conventions
  - All kinds of patterns
  - All kinds of pitfalls/anti-patterns
- Why is that the case?
  - Assume, you have implemented your own solution for a given problem
  - After a while, you find that a pattern exists the solves exactly the same problem
  - It is very likely, the pattern offers a better solution. Thus, consider refactoring your code to instantiate the pattern
Testing Refactorings

- To check the correctness of refactorings, various options are available
  - **Formal approach**: prove semantics and correctness of program transformation
  - **Implementation approach**: leverage unit tests to verify that the resulting implementation still meets the specification
  - **Architecture analysis**: check the resulting software architecture for its equivalence with the initial architecture (consider requirements)

- Use at least one verification method to ensure quality

Reversing Refactorings

- Refactoring represents a semantic preserving (i.e. structural) transformation that can be reversed
- For example, if you rename an entity you can also undo this refactoring
- Thus, refactoring patterns can be understood and applied in opposite direction
- Obviously, a refactoring should always be applied in the direction that improves architectural quality
- Note: That doesn't imply that a sequence of refactorings might always be reversed in deliberate order
Architecture Smells

- What are typical architecture smells that indicate we should refactor our architecture?
  - Breaking the DRY rule
  - Unclear roles of entities
  - Inexpressive or complex architecture
  - Everything centralized
  - „Not invented here“ syndrome
  - Over-generic design
  - Asymmetric structure or behavior
  - Dependency Cycles
  - Design violations (such as relaxed layering)
  - Inadequate partitioning
  - Unnecessary dependencies
  - Missing orthogonality
Rename Entities (1)

- **Context**
  - Using non intuitive names

- **Problem**
  - Your software architecture contains entities named in such a way that the whole architecture lacks expressiveness

- **General Solution Idea**
  - Introduce intuitive names so that stakeholders can easily understand the role of each subsystem
  - Change the names of the entities and also consider references to these entities
  - Change only one entity name at a time
  - Use one predefined naming scheme/strategy throughout the project

Rename Entities (2)

- **Example:**

  ![Diagram](Image)
Remove Duplicates (1)

- **Context**
  - Equivalent activities or artifacts that are duplicated
- **Problem**
  - DRY (Don’t Repeat Yourself) is an essential means to increase simplicity, expressiveness, orthogonality and reuse
  - If the same design artifacts are repeatedly implemented and used in a software architecture, then manageability and productivity will suffer
  - How can we prevent the introduction of equivalent design artifacts?
- **General Solution Idea**
  - Identify common (sequences of) tasks or otherwise related entities repeatedly used throughout your software system
  - Analyse if these common (sequences of) tasks or otherwise related entities can be provided as a generic component
  - If that’s the case, remove duplicates and introduce one common design artifact instead

Remove Duplicates (2)

- **Example**
  - **Client A**
    ```java
    main() {
      ...
    }
    ```
    Extract:
    - Connect JNDI
    - Get Home
    - Locate bean X
  - **Client B**
    ```java
    main() {
      ...
    }
    ```
    Extract:
    - Connect JNDI
    - Get Home
    - Locate bean Y
  - **Client A’**
    ```java
    main() {
      ...
    }
    ```
    Extract:
    call locateBean(X)
  - **Client B’**
    ```java
    main() {
      ...
    }
    ```
    Extract:
    call locateBean(Y)
Introduce Abstraction Hierarchies (1)

- **Context**
  - Entities representing related concepts

- **Problem**
  - In the architecture design entities appear that implement almost the same functionality
  - This leads to design and code replication as well as to less expressiveness and simplicity

- **General Solution Idea**
  - Leverage the Liskov Substitution Principle:
    - Introduce general abstractions containing the common parts of these entities
    - Define hierarchy of abstractions
    - Derive entities from the most specific abstraction
  - Different possibilities
    - Adding supertypes
    - Adding common interfaces
    - Adding common mix-ins, traits

Introduce Abstraction Hierarchies (2)

- **Example: Class Hierarchy**

![Class Hierarchy Diagram]

- Separate Abstractions
- Related Abstractions
- is-a Relation

© 2008 by Michael Stal
Remove unnecessary Abstractions (1)

- **Context**
  - Eliminating unnecessary design abstractions

- **Problem**
  - Minimalism is an important goal of software architecture, because minimalism supports quality attributes such as simplicity
  - The software architecture contains unnecessary abstractions that could be replaced by other already existing abstractions
  - How to get rid of unnecessary abstractions?

- **General Solution Idea**
  - Determine whether abstractions / design artifacts exist that could also be derived from other abstractions
  - If this is the case, remove superfluous abstractions and derive all dependent artifacts from other existing abstractions
  - Challenge: Don’t generalize too much (such as introducing one single hierarchy level, e.g., „all classes are directly derived from Object“)

Remove unnecessary Abstractions (2)

- **Example:**

![Diagram of abstract and concrete strategies involving storage and equipment with hierarchy levels and relationships.]
Substitute Mediation with Adaptation (1)

- **Context**
  - Centralized system using a mediation level between peers
- **Problem**
  - When interaction between different peers is complex, a mediator may be the right solution
  - However, extensive use of mediators may reduce scalability
  - If mediation was just applied to build an extremely generic solution, design erosion is often the price
  - How can we improve this situation?
- **General Solution Idea**
  - Introduce adapters for uniformly plugging in peers
  - but make interaction explicit, i.e., subsystems themselves are in charge to interact with the appropriate peers using an integration layer in an end-to-end fashion
  - Known Use: Enterprise Application Integration (Hub and Spoke) => Enterprise Service Bus

Substitute Mediation with Adaptation (2)

- **Example**

  ![Diagram](image)
Break Dependency Cycles (1)

- **Context**
  - Subsystem dependency cycles

- **Problem**
  - Your system reveals at least one dependency cycle in the subsystem relationship graph
  - A may either depend directly on B or indirectly (e.g., A depends on C which depends on B) which is why we always consider the transitive hull
  - Dependency cycles make systems less maintainable, changeable, re-usable, testable, understandable

- **General Solution Idea**
  - Get rid of any dependency cycle by introducing additional subsystems or breaking the cycle using inverse dependencies

Break Dependency Cycles (2)

- **Example:**

  ```java
  Sensor
  GetState()
  SetState()
  GetState()

  Monitor
  GetState()
  Draw()
  GetState()

  Date
  GetState()

  Break Cycle
  ```

  ```java
  Sensor
  GetState()
  SetState()
  GetState()

  Monitor
  GetState()
  Draw()
  GetState()

  Date
  GetState()
  ```

© 2008 by Michael Stal
Break Dependency Cycles (3)

- Variant: Dependency Inversion (e.g., switching push & pull)
- Example for Dependency Inversion:

```
interface IObserver {
    void Handle();
}

class Obs implements IObserver {
    void Handle() {
    }
}

class Subject {
    void Attach(IObserver ob) {
    }
    void Notify() {
        ob.Handle();
    }
}

class Subject {
    void Attach(IObserver ob) {
    }
    void Notify() {
        ob.Handle();
    }
}
```

Relationships between Refactorings

- When refactorings represent transformation patterns, we can also define
  - relations between refactorings
  - refactoring sequences, languages, systems, ...
  - different categories of architecture refactorings

- For example, an architecture refactoring such as Substitute Mediation with Adaptation may leverage smaller grained refactorings such as Integrate and Interoperate
Architecture and Code Refactorings

- If implementation is not yet available, architecture refactorings can be applied without any regard to implementation.
- If architecture refactoring is applied on architecture that already is connected to implementation:
  - Architecture Refactorings may trigger appropriate Code Refactorings.
  - Example: Architecture refactoring Rename Entity will cause appropriate code refactorings such as Rename Class/Method.
  - Thus, it is recommendable to document impact of architecture refactoring by specifying required code refactorings.
- Refactorings may overlap. Hence, all refactorings should be processed, one by one, ordered by priorities.
- Architecture refactoring before code refactoring.
- Prioritization of requirements suggests prioritization of refactorings.
- Strategic refactorings before tactical refactorings.
- Note: there is even overlap between some architecture and code refactoring patterns.

Other Types of Refactorings

- While the tutorial focused on architecture and code refactoring, there might be other refactorings as well.
- My brainstorming list:
  - Idiomatic Refactoring – how to more effectively use idioms of a programming language. Challenge: highly dependent on concrete language.
  - Test Refactoring – how to change a test plan. Challenge: consideration of dependencies on code and architecture refactorings.
  - Document and Diagram Refactoring – how to re-arrange a document or diagram to make it more readable and expressive. Challenge: avoid breaking of cross references, changing diagrams appropriately, layout.
  - Process Refactoring – how to change a process to make it more effective. For example: process improvement. Challenge: complexity due to activities, stakeholders, process interdependencies and interactions.
  - Organization Refactoring – how to change an organization to make it more effective. Challenge: often heavily dependent on processes.
  - Language Refactoring – how to change a language (e.g., a DSL) to make it more expressive.
Tool Support

- As Grady Booch once said „a fool with a tool is still a fool“
- However: A genius without a tool is like a mule
- Tool support for refactorings is essential to
  - avoid errors
  - check for consistency
  - provide what-if simulations
  - provide and manage key refactorings to engineers
  - increase efficiency

Summary and Wrap-Up (1)

- Refactoring changes the code/design without changing behavior or semantics. Refactoring helps for quality improvement or necessary adaptions
- In contrast to refactoring, reengineering is a complete redesign of an architecture and might also change behavior/semantics
- Both methods are essential, but use the right one for the right purpose
- Refactoring is not constrained to code but is applicable to architecture, test, documents, i.e. to all development artifacts
- Refactorings can be considered as transformation patterns (such as the optimization processing in a compiler)
- Thus, refactorings are discovered, not invented. Use best practices and pitfalls as potential sources
- Different flavors of refactorings are related. For example, architecture refactorings might imply code refactorings
Summary and Wrap-Up (2)

- If refactoring is applied, make sure your environment is appropriate
  - Development process should be agile (allow to design a little, code a little, test a little, ...)
  - Testing and Architecture Review are extremely important
  - (Code) Refactoring without appropriate tools is tedious and error-prone
- Become a discoverer and document refactorings you encounter in your daily engineering adventures
- Use Writer Workshops as a means to assure quality of refactorings
- Mind the power of refactoring: an innocently looking but wrong or not understood refactoring may cause a lot of harm

A departing thought

Each problem that I solved became a rule which served afterwards to solve other problems.

[René Descartes, 1596–1650, in "Discours de la Methode"]
References


- S. Demeyer, S. Ducasse, O. Nierstrasz: Object-Oriented Reengineering Patterns, Morgan Kaufmann, 2002

References (cont’d)

- Kerievsky, Joshua: Refactoring to Patterns, Addison-Wesley, 2004

References (cont’d)


Web References

- Martin Fowler, „Refactoring Home Page“
  http://www.refactoring.com
- Joshua Kerievsky: „Refactoring to Patterns Home Page“
  http://industriallogic.com/rtpdata/index.html
- Scott Ambler: “Database Refactoring”
  http://databaserefactoring.com
- Michael Stal, Software Architecture Blog
  http://stal.blogspot.com
Additional Architecture Refactorings

- Catalog

Inject Dependencies (1)

- Context
  - Components that need to create/locate other components

- Problem
  - If a component c needs access to other components, it most somehow create these components or locate them
  - However, this makes c to dependent on the knowledge about specific component location or factory mechanisms
  - How can we avoid these additional dependencies?

- General Solution Idea
  - Move functionality for discovery and creation of components to separate component m
  - Introduce configuration to let component c specify its requirements and let m be responsible for creating, discovering and then injecting required component references to c
Inject Dependencies (2)

- Example:

  ```java
  Component c { 
  S s = new S(.....); 
  s.m(); 
  }
  ```

  ```java
  Component c { 
  inject(S p) { s=p; }
  s.m(); 
  }
  ```

  ```java
  Component s = new C();
  S s = new S(.....);
  c.inject(s);
  ```

Insert Transparency Layer (1)

- Context
  - Decoupling clients from subsystems, and vice-versa

- Problem
  - When a client accesses subsystems directly, some dependencies are introduced, e.g.,
    - on the exact physical location of the subsystems, or,
    - on specific data or document formats the subsystem expects
  - The client must have knowledge about how to locate, access the subsystems
  - Finally, the client also becomes dependent on subsystem implementation aspects

- General Solution Idea
  - To avoid such dependencies, we introduce transparency layers to decouple these dependencies from clients
  - Transparency layers also open additional optimization possibilities
**Insert Transparency Layer (2)**

- Example:

  ![Diagram of Client to Subsystem B to Client](slide43.png)

  - This is a whole class of architecture refactorings, because for the implementation of transparency layers we could rely on patterns such as
    - Proxy,
    - Facade,
    - Wrapper-Facade,
    - Business Delegate,
    - Service Locator

---

**Reduce Dependencies with Facades (1)**

- **Context**
  - Client workflows that access multiple remote components

- **Problem**
  - If a client needs to access different external components for a workflow, it becomes dependent on details such as the set of available components. As soon as the configuration and/or interfaces of these components change, there will be a direct impact on the client
  - Performance might be an issue due to network roundtrips

- **General Solution Idea**
  - Introduce Facade component that acts as the client’s gateway into the set of required remote components
  - Implement workflow methods within the facade that represent the different workflows
  - The facade methods take over the responsibility to access the set of required components on behalf of the client, thus decoupling the client from the components
  - If components are remote, performance is also improved (Session Facade)
Reduce Dependencies with Facades (2)

- Example

```
Client ➔ Hotel Booking
   |               |
   | 1.1           |
   | Flight Booking|
   |               |
   |               |
   | 1.2           |
   | Rental Car Booking|
   |               |
   |               |
   | 1.3           |
   |               |
   | Facade for Travel Booking|
   |               |
   | 2.1           |
   | Flight Booking|
   |               |
   |               |
   | 2.2           |
   | Hotel Booking|
   |               |
   |               |
   | 2.3           |
   | Rental Car Booking|
   |               |
   |               |
   |               |
   |               |
   |               |
   |               |

Client ➔ Facade for Travel Booking
```

Replace Implicit with Explicit Workflows (1)

- Context
  - Implementing and maintaining workflows

- Problem
  - Workflows within a system may be built using a programming language such as Java
  - However, when workflows need to be changed often, this requires direct and frequent modification of hard-coded workflows
  - How can we support workflows that are easy to change and modify, even by domain experts?

- General Solution Idea
  - Define a (domain-specific) workflow language or use an existing one
  - Build a workflow runtime to which you may pass workflow descriptions
  - Let the runtime execute these descriptions
  - Provide a callback strategy so that the workflow engine may inform your application about relevant (lifecycle) events
Replace Implicit with Explicit Workflows (2)

- **Example**

  ```java
  loop {
  decimal d = Stocks.get(SUN);
  if (d >= 8.5) webserviceSell();
  else webserviceBuy(…);
  if (endOfTrading()) break;
  }
  ```

- **See also:** Integrate DSLs, Object Manager, and Reduce Dependencies with Facades

Move Entities (1)

- **Context**
  - Moving entities between subsystems

- **Problem**
  - This can be considered a generalization of the Split Subsystems refactoring
  - In a subsystem an entity is defined that semantically better fits to another subsystem to which it has high coupling

- **General Solution Idea**
  - Again, we should consider cohesion and coupling
  - If entity e reveals low or zero cohesion to the rest of its subsystem A, but tight coupling to subsystem B, then move e from A to B
  - Note: This may also help for Breaking Dependency Cycles between subsystems
Move Entities (2)

- Example:

```
Game
getPlayerHistory(p)

Player
```


Move Entities (3)

- Some additional remarks
  - In many cases we can't constrain the refactoring to single atomic entities such as a class, method, constant, interface
  - More likely, there will be clusters of atomic entities that together build a complex entity
  - In the example, it is very likely that there will be classes and interfaces related with the method to be moved
  - Thus, always consider semantically related entities with high internal cohesion but less cohesion to the rest of their subsystem. If some of them show tight coupling to another subsystem, move the whole group
  - Moving may not always be a simple operation, but imply several smaller grained transformations (see example)
Merge Subsystems (1)

- **Context**
  - High Coupling between subsystems

- **Problem**
  - Between two subsystems in a software architecture, the degree of coupling should be rather loose
  - Within a subsystem the number of interdependencies (cohesion) should be high
  - If the coupling is too tight, then the many interdependencies between these two subsystems decrease qualities such as
    - Changeability
    - Performance (maybe)
    - Re-Usability of one of the subsystems

- **General Solution Idea**
  - Tight coupling between subsystems implies that the two subsystems in fact implement one subsystem
  - Either merge both subsystems to one, or,
  - Move functionality from one subsystem to the other

Merge Subsystems (2)

- Example: While tight coupling between subsystems is bad, high cohesion within subsystems is good. Thus, merge tightly coupled subsystems to form a highly cohesive subsystem

- Special variant: merge layers in a layered system
Split Subsystems (1)

- **Context**
  - Low cohesion within a subsystem

- **Problem**
  - Within a subsystem the interdependencies (cohesion) should be high
  - Between two subsystems in a software architecture, the degree of coupling should be rather loose
  - If the cohesion between some parts is loose, then some design decisions seem to be questionable
  - It is recommendable to change this to obtain better modularization and understandability
  - Another potential problem are subsystems/components with too many responsibilities

- **General Solution Idea**
  - Loose cohesion within a subsystem implies that the functionality can be split into multiple subsystems
  - Thus, determine areas with high cohesion in a subsystem. All those areas with low cohesion are candidates for becoming subsystems of their own

Split Subsystems (2)

- **Example:** When analysing interdependencies between entities in a subsystem two (or more) sets with high cohesion can be determined. Within these sets there is high cohesion, between these sets there is only low cohesion. Thus, split the subsystem into two (or more) parts

Special variant: Split layer in a layered system
Enforce Strict Layering (1)

- **Context**
  - An architecture with accidentally broken layering
- **Problem**
  - Layers are introduced to reduce dependencies, increase adaptability and re-usability, as well as maintainability
  - Sometimes, relaxed layering is used intentionally to solve problems such as performance issues
  - Other times, relaxed layering is used accidentally without any justification and value
  - In the latter case, how can we cure the broken layering?
- **General Solution Idea**
  - Re-design so that the broken layering is substituted with strict layering

Enforce Strict Layering (2)

- **Example:**

![Diagram of strict layering](image-url)
### Enforce Strict Layering (3)

- Unfortunately, broken layering is often hard to cure
- To resolve the broken layering between component 3.1 and component 1.1, we must differentiate the following cases:
  - Case 1: all the kinds of functionality of component 1.1 that component 3.1 uses, are also available in layer 2
    - In component 3.1 substitute all usages of component 1.1 by equivalent functionality of layer 2
  - Case 2: not all kinds of functionality of component 1.1 that component 3.1 uses, are also available in layer 2
    - Where possible, substitute all those component 1.1 invocations with equivalent invocations of layer 2. For the rest of invocations move functionality from layer 3 to layer 2
  - Sometimes, it is not recommendable in case 2 to move functionality from layer 3 to layer 2. Then, consider re-engineering activities instead!
  - Remark: This is one of the refactorings that can also be applied in the opposite direction. If strict layering is not feasible (e.g., due to performance reasons) you might want to relax the strict layering!

### Add Strategies (1)

**Context**
- Enable selection and exchange of implementations

**Problem**
- Services and algorithms in a component can be implemented in various ways. Their implementations often also depend on client requirements
- The client should not depend on the concrete implementations used
- Hardcoding a service or algorithm and then exchanging it at source code level is a possible solution
- Unfortunately, this does not hold for run time configurability
- How can we refactor a component to support this kind of run time configuration?

**General Solution Idea**
- Use the Strategy pattern
Add Strategies (2)

- Example

```
Client
  ▼
Middleware
    ▼
Request Dispatcher
      ▼
Protocol
```

```
Client
  ▼
Middleware
    ▼
DispatcherStrategy
      ▼
ConcreteDispatcher
      ▼
Protocol
```

- Warning
  - Do never introduce strategies in an uncontrolled way
  - A strategy basically means I have no idea which option to use at run time
  - Strategy Syndrome is a serious disease that leads to an over-generic architecture which is doomed to design erosion

Enforce Symmetry (1)

- Context
  - Refactor an asymmetric solution

- Problem
  - Asymmetry reduces conceptual integrity and simplicity
  - Structural asymmetry deals with the usage of the same solution concepts throughout an architecture
  - Functional asymmetry is relevant on the method level ("when there is an open, there should be a close operation")

- General Solution Idea
  - Prioritize strategy over tactics
  - For identical problem contexts in your architecture apply the same solutions (e.g., the same patterns)
  - Use refactoring-to-patterns where applicable
  - All unmotivated asymmetries should be removed step-by-step
Enforce Symmetry (2)

- Examples for structural asymmetry:
  - Some places in the architecture throw exceptions, while other return error codes
  - One subsystem leverages the Observer pattern, another one a hardwired event notification strategy

Enforce Symmetry (3)

- Example for functional asymmetry: Abstract Factory with missing dispose method:
Extract Interface (1)

- **Context**
  - (Sub)system needs to export functionality

- **Problem**
  - An existing (sub)system needs to export some of its internal functionality to external users
  - How can the (sub)system developer make internal functionality available externally?

- **General Solution Idea**
  - Define abstract interface and contract
  - Provide wiring of interface methods to internal functionality
  - Apply patterns such as Bridge, Decorator or Adapter to separate interface from implementation
  - Optionally, use Extension Interface pattern to manage multiple interfaces

Extract Interface (2)

- **Example**

  ```java
  // Sensor
  getTemperature(UNIT)
  getAverage(from, to)
  storeRecord()
  getRecord(DateTime)
  getDateTime()

  // Adapter
  if (UNIT == CELSIUS)
      useCelsius();
  else
      useFahrenheit();
  return temperature();
  ```

- **Warning:** Only export those internal methods that are required externally! Otherwise, you might introduce unnecessary dependencies or security holes.
Enforce Contract (1)

- **Context**
  - Enforcing contract of an interface

- **Problem**
  - A contract defines what the user of an interface must provide and what the user can expect in return, e.g., preconditions, postconditions, required order of method execution
  - Often, interface implementations do not enforce the contract
  - A contract that is never enforced is of limited value
  - How can we enforce the interface contract?

- **General Solution Idea**
  - Define contract explicitly: for instance, use a state machine to define method execution order. Specify interface invariants. For each method, determine preconditions and postconditions.
  - Enrich interface implementation to enforce contract

Enforce Contract (2)

- **Example**

```
<< interface >>
IFile
Handle open(…)
write(…)
close(…)
```

```
<< interface >>
IFile
Handle open(…)
write(…)
close(…)
```

Contract Implementation

```
Open/isValid(File)
Write/notEmpty(buf)
Close
```

File System
**Introduce Extension Interfaces (1)**

- **Context**
  - Bloating of component interfaces
- **Problem**
  - If a component evolves over time, often new functionality is added to the component's interfaces
  - This may lead to severe interface bloating which has a negative impact on quality attributes such as usability and manageability (Swiss Army Knife anti pattern)
  - Unsystematic interface evolution may also break client code
  - How can we add an interface management for systematically evolving and providing interfaces?
- **General Solution Idea**
  - Apply Extension Interface Patterns [POSA2]:
    - Introduce a common protocol for all provided interfaces (incl. interface navigation)
    - Integrate additional functionality so that clients can discover existing component interfaces and navigate between them

**Introduce Extension Interfaces (2)**

```plaintext
Factory
create()
find()

RootInterface
get_extension()

Interface1
service_1.1();
service_1.2();
service_1.3();
service_1.4();
.
.
.
.
.
.
.
.
.
service_1.n();

Component

Extension Interface1
service_1();

Component

Extension Interface2
service_2();

Client
```

© 2008 by Michael Stal
**Introduce Extension Interfaces (3)**

- **Notes**
  - This refactoring alters components and clients. Thus, handle with care!
  - Interface navigation should be reflexive, symmetric, transitive
  - But you may also introduce a dedicated interface that includes the navigation functionality

![Diagram of Introduce Extension Interfaces](slide69.png)

**Substitute Inheritance with Delegation (1)**

- **Context**
  - Using delegation when inheritance is not available or not suitable
- **Problem**
  - Assume an entity $D$ should logically inherit from entity $B$, but inheritance is not available on component or subsystem level, or inheritance can not be used due to other restrictions (e.g., single inheritance)
  - In these cases using inheritance is not the right approach
  - How can we get rid of (the necessity to use) inheritance without losing all its capabilities?
- **General Solution Idea**
  - Keep $B$ as it is
  - Within entity $D$ add reference to $B$
  - Provide public interface of $B$ also in $D$ but delegate all invocations to referred $B$ (optionally passing a reference to $D$ for back calls)
- **Remark**
  - This is another example of the reversible patterns that may also be applicable in the opposite direction!
Substitute Inheritance with Delegation (2)

• Example

```java
IPlayer
play()

Player
play(IPlayer)

play()

Ninja
lookupExperience()

Player
play(IPlayer)

play()

Ninja
lookupExperience()

Player
play(IPlayer)

play()

IPlayer
play()
```

Provide Interoperability Layers (1)

• Context
  • Enable interoperability between two subsystems

• Problem
  • Sometimes, formerly collocated subsystems need to be distributed
  • Or, an application needs to interoperate with other legacy applications
  • Unfortunately, both of them were not prepared to interoperate

• General Solution Idea
  • Insert adapter layers to provide interoperability
  • Adapt subsystem A to B or subsystem B to A or use a bidirectional adaptation
  • Adaptation is a holistic approach. You may adapt communication, data, policies
  • Use patterns such as Adapter or Wrapper Facade. In other words apply the Insert Transparency Layer refactoring
Provide Interoperability Layers (2)

- **Example**
  - Provide bi-directional interoperability between two subsystems using Web Services
  - Extract subsystem interfaces and use adapters (e.g., session facades, business delegates) to integrate them with their subsystems

Provide Interoperability Layers (3)

- **Aspects for interoperability in object-oriented, distributed systems:**
  - Interoperability is not restricted to data transformation and communication
  - For instance, we also need to adapt error management.
  - Likewise, object models must be mapped onto each other.
  - Policies (e.g., security, memory management) must be mapped onto each other
  - The simpler the adaptation layers provided, the more work developers must spend to close the semantic gap
  - Integration products such as EAI solutions just provide a set of proprietary interoperability solutions
Aspectify (1)

- **Context**
  - Dealing with cross-cutting concerns
- **Problem**
  - Cross-cutting concerns such as security mechanisms, distribution mechanisms and other invasive functionalities lead to extensive replication of design and code
  - OO mechanisms such as inheritance can’t solve the problem due to infeasible centralization of these concerns
  - Nonetheless, separation of concerns should be supported
- **General Solution Idea**
  - Express concern as a centralized package
  - Introduce injection mechanism to integrate wrapped concern in all required places of software architecture
  - If cross-cutting concerns are interdependent, allow injection mechanism to take care of interdependencies, for example by using priorization or generative approach

Aspectify (2)

- **Example:**

![Diagram showing an example of cross-cutting concerns and how they can be handled with the Aspectify solution.](image)
Integrate DSLs (1)

- **Context**
  - A system that covers many different domains

- **Problem**
  - Mastering complexity in a software system is an important challenge. Complexity is often caused by the use of multiple domains (based on different concepts and paradigms)
  - If many domains are involved such as the business logic itself, accessing databases, specifying workflows, UI design, then impedance mismatch becomes an essential issue
  - In addition, work of different domain experts is more difficult because all domain experts must understand the full software system

- **General Solution Idea**
  - Instead of hard-coding domain-specific functionality manually, define a metamodel/DSL that allows domain experts to define their logic
  - Implement generators that transform models/DSLs to implementation artifacts
  - Define an integration subsystem for each of these domains that integrates the generated artifacts with the rest of the system

Integrate DSLs (2)

- **Example**
  - Microsoft’s XAML (eXtensible Application Markup Language) is used for declaratively specifying workflows within and between applications
  - In the past, developers had to hardcode these workflows in the architecture using implementation languages such as C#
Add Uniform Support for Runtime Aspects (1)

• Context
  • Software architecture that must support runtime aspects such as lifecycle management, management, configuration

• Problem
  • Each non-trivial software architecture consists of multiple components each of which supports common aspects
  • If every component implements its own interfaces for runtime aspects, the overall system will lack simplicity and expressiveness
  • Is there a way to provide a uniform approach to support those kinds of runtime aspects

• General Solution Idea
  • For each aspect, define a common interface (e.g., an interface for runtime configuration)
  • In each component supporting the aspect provide this common interface (maybe, using Extension Interfaces to prevent bloating)
  • To achieve orthogonality, provide one runtime component in charge of accessing these common interfaces (e.g., a management console)
Add Configuration Subsystem (1)

- **Context**
  - A system with many configurable variabilities

- **Problem**
  - If a system contains a lot of configuration options, then configuration itself is often tedious and error-prone
  - This holds in particular, when the configuration options are (partially) related with each other
  - How can we refactor the software system so that configuration is simplified and (partially) automated?

- **General Solution Idea**
  - Instead of providing dozens of configuration options to external actors, we introduce a configuration subsystem that takes a declarative configuration (from a file or a wizard)
  - The configuration subsystem reacts on configuration change events, reads the passed configuration and then accesses the configuration interfaces of configurable subsystems
  - Ideally, all subsystems provide the same generic configuration interface

Add Configuration Subsystem (2)

- **Example**

  ![Diagram](https://via.placeholder.com/150)

- **Use patterns such as Component Configurator for that purpose**
Open your Architecture (1)

- **Context**
  - Opening a subsystem for external monitoring or modification (e.g., eventing or interception)

- **Problem**
  - An existing subsystem needs to be opened to let other subsystems observe this subsystem or intercept specific behavior
  - So far, the subsystem has been a closed silo
  - How can we introduce mechanisms for opening our architecture in a controlled way?

- **General Solution Idea**
  - Apply the Open/Close Principle:
    - Design the states and state transitions of the subsystem and decide which transitions should be observable or interceptable
    - For all these observation and interception points introduce appropriate hooks with which other subsystems can register for observation or interception
    - If interception is required, design the required context object and subsystem internal functionality required for interception

Open your Architecture (2)

- **Example:**

```
Sensor
ObserverInterface
SensorMonitor
```
Optimize with Caching (1)

- **Context**
  - Optimize access to re-usable resources using caching

- **Problem**
  - When a subsystem provides resources to other parties, resource access becomes a potential bottleneck
  - If high usage frequency of the same resources leads to performance bottlenecks due to costs for resource creation, allocation, initialization addition means for accelerating resource access are necessary
  - How can we optimize resource access?

- **General Solution Idea**
  - Introduce a fast cache where re-usable resources are maintained
  - Define an appropriate resource allocation (activation) and a deallocation (eviction) strategy
  - Provide a transparency layer to resource users

Optimize with Caching (2)

- **Example:**

```plaintext
Client

Subsystem
getResource()
allocates & creates
Resource

Subsystem
getResource()
Cache
acquire()
release()
maintains
Resource

Client
```
Optimize with Caching (3)

- The choice of allocation/deallocation strategy is important
- Example
  - A web site that returns weather maps based on zip codes
  - Suppose, zip-based maps and weather data are stored in databases.
    When a new request arrives
    - a servlet reads the map
    - reads the weather data,
    - and then uses image processing to calculate the map with weather information
    - which it finally returns to the client
  - When we introduce a cache how do we decide which maps to store?
    Should we always remove the least frequently used maps? Or could we just store the „empty“ map in the cache?
  - How should we deal with updates (weather changes)? For example, can we introduce timeout values (leasing)?

Replace Singleton [1st Variant] (1)

- Context
  - Resources/objects with state that should be available at most once
- Problem
  - Assume, a resource type (objects within a class, threads in a thread pool, …) should only contain one instance w.r.t. its state
  - For this purpose, the Singleton Pattern is often applied
  - Unfortunately, the Singleton pattern acts like a global declaration
  - Are there any alternatives for our problem context?
- General Solution Idea
  - In a singleton, state and behavior are integrated. The non-transient parts of the state represent the identity of the singleton object
  - Thus, the singleton mustn’t be replicated as this would duplicate its identity
  - Replace the singleton class with a class that provides the same public interface (value class), but store all the non-transient state in a central place (database, TSS, hash table, …) with the key known to the new class. The behavior and transient state, however, may be replicated
Replace Singleton [1st Variant] (2)

- **Example**

- State is now a singleton while behavior isn’t
- In the end it’s all about identity
- Known Use: EJB Entity Beans

```
+ m1();
+ m2();
+ m3();
state
```

Replace Singleton [1st Variant] (3)

- **Virtual Objects:**
  - The same concept is applicable to introduce operational qualities such as availability or fault-tolerance through replication
  - In this case a number of objects provide the same services but store their state in a central place
  - For clients all objects appear as one single (virtual) object
  - For state synchronization between physical objects (that represent the same virtual object), a protocol is required
  - Patterns to be applied:
    - Half-Sync-Plus-Protocol
    - Object Group

```
HalfObject1
service1()
service2()
protocol()

HalfObject2
service1()
service3()
protocol()

HalfObjects
Here
local
remote

There
Protocol
```
Replace Singleton [2nd Variant]

- Another possibility to remove singletons is introducing a wrapper such as an object manager or factory.
- The wrapper is in control of lifecycle management and thus able to control that a specific type has only one instance.

```
Singleton
+ m1();
+ m2();
+ m3();
state
Wrapper
```

The Wrapper enforces that at most one instance of the Singleton is created. Note: a singleton might be removed and then reactivated, e.g., due to leasing.

Separate Synchronous and Asynchronous Processing (1)

- **Context**
  - Distributed systems including a mixture of synchronous and asynchronous processing

- **Problem**
  - Both asynchronous and synchronous services should be decoupled so that neither suffers from deficiencies of the other.
  - Both asynchronous and services should be able to communicate with one another efficiently using their own communication paradigm.
  - How can we refactor an architecture where synchronous and asynchronous services are tightly coupled?

- **Solution**
  - Separate synchronous and asynchronous services from another by dedicated layers and add a queuing layer between them to mediate communication between services (Half Sync/Half Async Pattern).
Separate Synchronous and Asynchronous Processing (2)

- Example

![Diagram showing synchronous and asynchronous service layers with examples of methods and queues.]

Replace Remote Methods with Messages (1)

- Context
  - Asynchronous communication in a distributed system

- Problem
  - In many distributed systems remote method invocation is the primary choice of communication. Asynchronous communication is introduced in the application layer but implemented using synchronous remote method calls.
  - However, message-oriented middleware is the better choice for dealing with event-based or asynchronous communication.
  - Is there a migration path to move from synchronous to asynchronous communication without huge impact?

- General Solution Idea
  - Keep the Broker-based architecture unchanged but integrate an additional asynchronous layer that relies on messaging middleware.
  - For synchronous communication still use the synchronous layer, but for asynchronous communication use the asynchronous layer instead (message queues, asynchronous completion tokens, ...).
  - Future functionality may then directly rely on asynchronous messaging.
Replace Remote Methods with Messages (2)

- Example

Client
  service_c()

C_Side_Proxy
  service_s()
  snd_request()
  rcv_response()

Broker
  est_connection()
  find_server()
  register_svt()
  unregister_svt()

Bridge
  est_connection()
  find_server()

Servant
  service_s()

S_Side_Proxy
  rcv_request()
  snd_response()

S_Side_Proxy
  service_s()
  snd_request()
  rcv_response()

Queue
  listen()
  enqueue()
  dequeue()

Add Object Manager (1)

- Context
  - Management support for objects

- Problem
  - Objects (and resources) for client usage often need to implement management functionality in addition to business logic, e.g., lifecycle management, persistence management
  - This burden leads to memory usage and responsibility overload
  - Clients shouldn’t be responsible for object management either
  - Different applications may require different management policies
  - How can we provide such an additional management layer?

- General Solution Idea
  - Separate object usage from object management
  - Introduce an manager component that manages objects throughout their lifetime
Add Object Manager (2)

- Example

```
Client
create()
delete()
find()
```

```
Factory
create()
delete()
find()
```

```
Object
service()
```

```
Object Manager
createObject()
deleteObject()
insertObject()
removeObject()
findObject()
objectIterator()
```

```
ManagedObject
service()
```

Change Unidirectional Association to Bidirectional (1) [Fowler]

- Context
  - Two classes referring to each other

- Problem
  - Class A refers to a class B, but there is no backward navigation from B to A
  - However, after some development activities, you recognize that B also requires a reference to all A objects referring to it
  - How can we transform the unidirectional to a bidirectional navigation?

- General Solution Idea
  - Add a back link in class B that allows to navigate from B to the referring A object(s)
  - Decide class that is in charge of controlling the association, e.g.,
    - The composite controls its constituents
    - In one-to-many relationships make the one-side the controller
  - Introduce helper method in noncontrolling side
  - If existing modifier is on controlling side, modify it to update back references, otherwise create a controlling method on the controlling side and call it from existing modifier
Change Unidirectional Association to Bidirectional (2) (Fowler)

- Example

```java
class Order {
    Customer getCustomer() {
        return _customer;
    }
    void setCustomer(Customer arg) {
        _customer = arg;
        Customer _customer;
    }
}

class Customer {
    private Set _orders = new HashSet();
    void addOrder(Order arg) {
        arg.SetCustomer(this);
    }
    Set friendOrders() {
        return _orders;
    }
    
    class Order {
        void setCustomer(Customer arg) {
            if (_customer != NULL)
                _customer.friendOrders().add(this);
        }
    }
}
```

Advanced Refactoring Issues

- Refactorings are not islands!