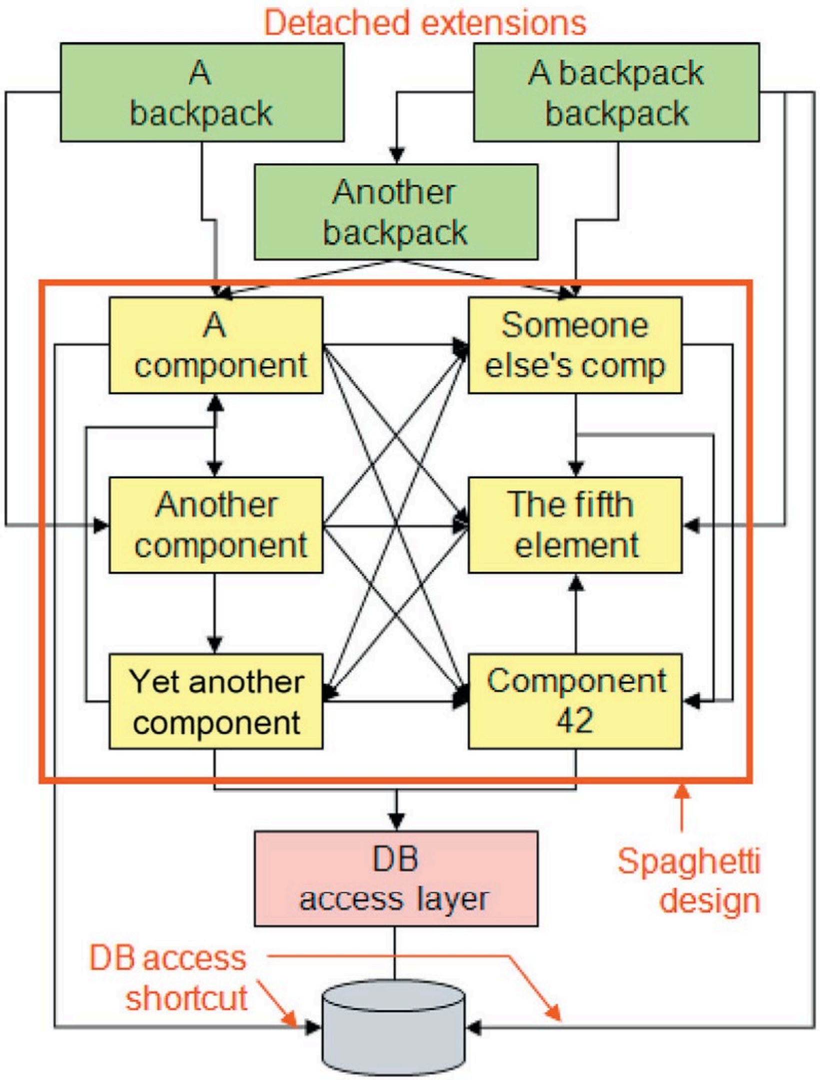
Architecture Refactoring

Software Evolution

- software systems continuously change
  - requirements addition of refinement, infrastructure or technology changes, bug fix, design decision change, …
- causes architectural erosion
  - unmanageable and unmaintainable system design
- requires systematic enforcement of **architecture assessment** and **refactoring** to every design activities

Design Erosion

- teleco. System
  - unsystematic architecture evolution – ad hoc patches and backpacks
  - high complexity
  - low modifiability
  - performance penalties
- **iterative** architecting
- architecture **evaluation** and **refactoring** in each iteration



Software Evolution & Refactoring

- **code refactoring** (Fowler)
  - changing source code without modifying its external behavior
  - key software engineering (agile) practice
- **refactoring to patterns** (Kerievsky)
  - substitute 'proprietary' solutions with design patterns
  - shifting focus from code to design
- **architecture refactoring**
  - no amount of code refactoring can replace huge benefits delivered from major structural clean up or change

Architectural Smells

- **duplicate design artifacts**
  - DRY (Don't Repeat Yourself) principle violation
  - same responsibility assign to different architecture components – common task should be modularized
- **unclear roles of entities**
  - SRP (single responsibility principle) or separation of concerns principle violation
  - responsibilities should be assigned to individual components – not spread across multiple components
- **inexpressive or complex architecture**
  - accidental complexity leads to unnecessary abstraction causing complex and inexpressive software
    - unclear or misleading names
    - superfluous components or dependencies
    - too fine or too coarse granularity

Architectural Smells

- **everything centralized**
  - centralized approaches when self organization and decentralization would be more appropriate
  - use decentralized approach, if problem is inherently decentralized
- **home-grown solution instead of best practices**
  - reinventing wheel instead of using proven solution
    - high probability that well known solutions (patterns) are superior
- **over-generic design**
  - architecture design should be as specific as possible and only as generic and configurable as necessary
    - overuse of strategy pattern (to defer variability to later binding time) makes maintainability and expressiveness suffers

Architectural Smells

- **asymmetric structure or behavior**
  - symmetry is an indicator for high internal architectural quality
    - behavioural : open - close, transaction – commit/rollback, fork - join
    - structural : solve identical problems using identical pattern
- **dependency cycles**
  - dependency cycles among architectural components indicates problems – might cause negative impact on testability, modifiability, expressiveness
- **design violations**
  - violation of design policies (e.g. use of relaxed layering)
    - different engineers might resolve the same kind of problem differently in uncontrolled way reducing visibility and expressiveness

Architectural Smells

- **inadequate partitioning of functionalities**
  - inadequate responsibility mappings to subsystem causing accidental complexity – cause low cohesion and high coupling
- **unnecessary dependencies**
  - minimizing dependencies reduce complexity
  - all additional and necessary dependencies might affect performance and modifiability
- **implicit dependencies**
  - if implementation contains dependencies not available in architectural models, it may cause many liabilities
  - changes without being aware of implicit dependencies can break implementation

Architectural Smells - Alternative

- **cyclic dependency**
  - 2 or more architecture components depend on each other directly or indirectly
- **unstable dependency**
  - when a component depends on other components that are less stable then itself
- **ambiguous interface**
  - when a component offers only a single, general entry-point into the components
- **god component**
  - when a component is excessively large (LOC or number of classes)
- **feature concentration**
  - when a component realize more than 1 architectural concern/feature
- **scattered functionality**
  - when multiple components are responsible for realizing the same high-;level concern
- **dense structure**
  - when components have excessive and dense dependencies without any particular structure

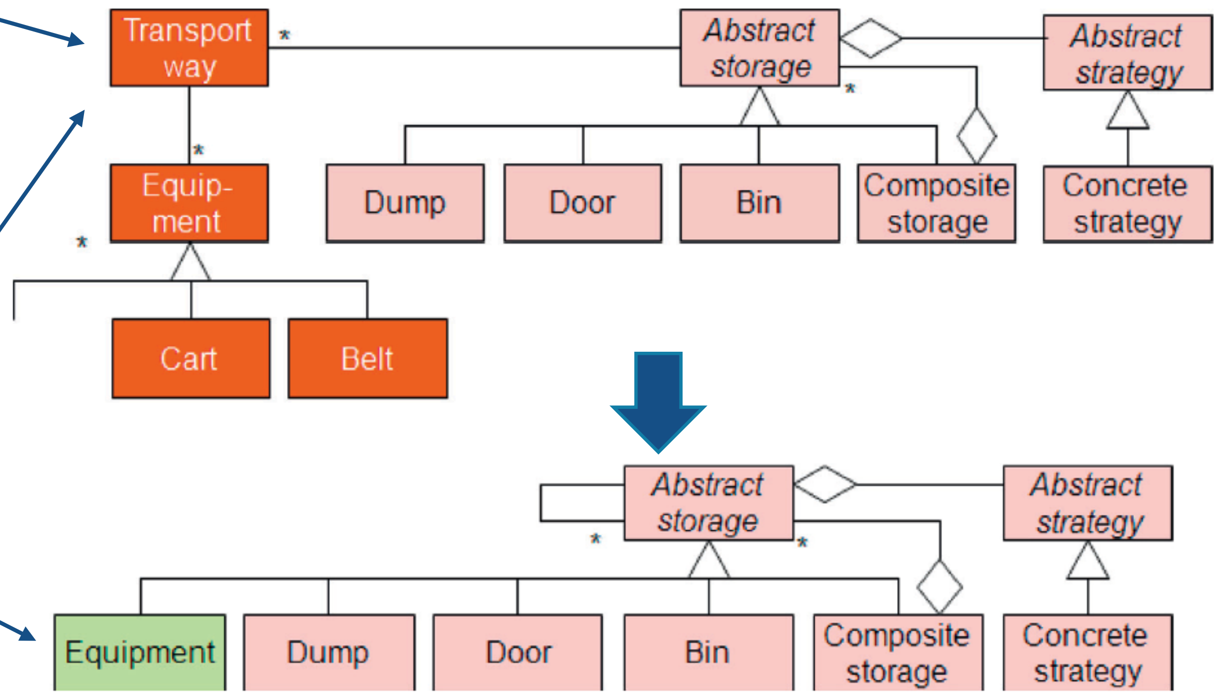
Refactoring Tactics

- example : warehouse management system

increase complexity with additional components and relationship

resembles Abstract storage with associated strategy defining their concrete transport kind

eliminate Transport way by considering Equipment as a special storage kind



  - configurability (configuring concrete storages) & changeability (exchanging concrete storage / strategy) is improved significantly

Refactoring Tactics

Name: Remove unnecessary abstractions in abstraction hierarchies

- Context
  - Removing unnecessary design abstractions after system extension
- Problem
  - Minimalism is an important goal of software architecture, because minimalism increases simplicity and expressiveness
  - If the software architecture comprises abstractions that could also be considered abstractions derived from other abstractions, then it is recommendable to remove these 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 dependent from existing abstractions
- Caveat
  - Don't generalize too much (such as introducing one single hierarchy level: "All classes are directly derived from Object")

Refactoring – Possible Example

- partition responsibilities:
  - component / subsystem with too many responsibilities
  - partition component / subsystem into multiple parts, each of which with semantically related functionality
- extract service:
  - a subsystem does not provide any interfaces to its environment but is subject of external integration
  - extract service interface
- introduce decoupling layer:
  - components directly depend on system details
  - introduce decoupling layer(s)

Refactoring – Possible Example

- rename entity:
  - entities got unintuitive names
  - introduce appropriate naming scheme
- break cycle:
  - when encountering a cycle on subsystem level
  - break it
- merge functionality:
  - if there is broad cohesion between two module

Refactoring – Possible Example

- orthogonalize:
  - two parts of an architecture introduce different solutions for the same problem
  - choose one preferred solution and eliminate the other
- introduce strict layering:
  - in a layered system, a layer accesses lower layers without necessity (relaxed layering)
  - enforce strict layering
- introduce hierarchies:
  - several entities are only variants of a particular entity
  - introduce a hierarchy

Refactoring – Possible Example

- introduce Interceptor hooks:
  - we have to open an architecture for out-of-band functionality according to the Open/Close principle interceptors should be introduced

- eliminate dependencies by dependency injection:
  - reduce direct and wide-spread dependencies of Parts in a Whole/Part setting by introducing a central runtime component (Whole´) that centralizes dependency handling with dependency injection

Quality Improvement

refactoring goal is to improve

- **internal quality**
  - **economy** : follow 'Keep it Simple, stupid!' (KiSS) principle for architecture to contain only required artifacts to achieve goals
  - **visibility** : all parts should be easily comprehensible and there should be no implicit components and dependencies
  - **spacing** : good separation of concern to efficiently and effectively map responsibilities to entities
  - **symmetry** : lack of symmetry indicates possible design problems
    - behavioural : open - close, transaction – commit/rollback, fork - join
    - structural : solve identical problems using identical pattern
  - **emergence** : rely on simple constituents with complex functionality than centralize the same functionality in complex and heavyweight artifacts
- **external quality** : quality attributes such as **ISO/IEC 25010**

Refactoring Process- Rough Outline

1. **architecture assessment**
   - identify architecture **smells** and design problems
     - use code quality management tool, architecture assessment tools , architecture review methods
   - create list of identified issues

2. **prioritization**
   - **prioritize** all identified architectural issues based on priority of the affected requirements
     - solve problems related to strategic design before addressing tactical areas
     - cover artifacts associated with high-priority requirements before ones with lower priorities

Refactoring Process- Rough Outline

3. **selection** – for each problems in the prioritised list :
   i. select appropriate **refactoring** pattern
   ii. if more than one pattern exists, choose the one that reveals appropriate consequences for the system
   iii. if no pattern exist, fall back to conventional architectural redesign

4. **quality assurance** – check for accidental semantic change :
   i. **formal methods** : prove that structural transformation did not change behavior
   ii. **architecture assessment** : check quality through architectural or design review (code review is valuable if implemented)
   iii. **testing** : if already implemented, use existing tests and also apply software quality metrics

Shallow & Deep Refactoring

- refactoring depend on whether the affected part of architecture is already implemented or not

- **shallow refactoring** - architecture as set of models (views)
  - refactoring only implies model refinement and modification
  - correctness is checked with architecture assessment methods

- **deep refactoring** – architecture with implementations
  - applying refactoring pattern will also require code refactoring
  - might also impact further artifacts - documentation, database schemas, reference architectures, …
  - additional quality assurance can be achieved by testing

Architecture Refactoring - Obstacles

- management and organization
  - PM considers new feature the most important
  - considers architecture should be done correctly in first place, so that no problems ever appears
- development process
  - refactoring process should be explicitly integrated – to plan sufficient resources
- technology and tools
  - lack of support tools – tedious and error-prone manual process
- applicability
  - with significant design erosion, reengineering or rewriting might be more appropriate and efficient