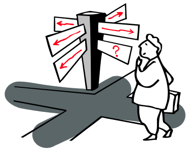
CHAPTER 8 – Software Architecture



Introduction
- When, Why and What?
- Functional vs. Non-functional
- Coupling and Cohesion
- Patterns

Macro architecture
- Layered Architecture
- Pipes and Filters
- Blackboard Architecture
- Model-View-Controller

Micro Architecture
- Observer
- Abstract Factory
- Adapter (a.k.a. Wrapper)
- Bridge & Facade

Conclusion
- Architecture in UML
- Architecture Assessment
  + ATAM
- Correctness & Traceability

Literature (1/2)

Software Engineering Text Books
- [Somm04a]: chapter “Architectural Design”
- [Pres01a]: chapter “Architectural Design”

Books on Software Architecture
- [Shaw96a] Software architecture: perspectives on an emerging discipline, Mary Shaw, David Garlan, Prentice-Hall, 1996.
  + The book introducing software architecture.
- [Bass98a] Software architecture in practice, Len Bass, Paul Clements, Rick Kazman, Addison-Wesley, 1998. (There is a 2nd edition)
  + A very deep and practical treatment of software architecture, *incl. ATAM*. (The book received an award.)
- [Bosc99a] Design and use of software architectures: adopting and evolving a product-line approach, Jan Bosch, Addison-Wesley, 2000.
  + How to build product-line architectures, including a number of cases.

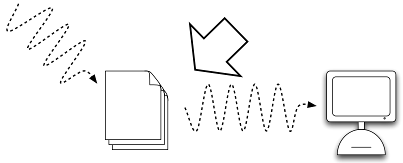
Articles
- Philippe Kruchten "The 4+1 View Model of Architecture ", IEEE Software, November 1995 (Vol. 12, No. 6) pp. 42-50.
  + A paper that illustrates convincingly the need for various perspectives on the design of a system.

Literature (2/2)

Pattern Language
- [Foot97a] Big Ball of Mud, Brian Foote, Joseph Yoder; Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97)
  + http://www.laputan.org/mud/mud.html; most popular architecture.

Pattern Catalogues
- [Busc98a] Pattern-Oriented Software Architecture: A System of Patterns, Frank Buschman, Regine Meunier, Hans Rohnert, Peter Somerlad, Michael Stal, Wiley and Sons, 1996.
  + Introduces architectural styles in pattern form. Also covers some design patterns and idioms.
    ➡ At architecture (= “macro-architecture”) level
- [Gamm95a] Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley, 1995.
  + The classic; commonly referred to as the “Gang of Four (GOF)”
    ➡ At design (= “micro-architecture”) level

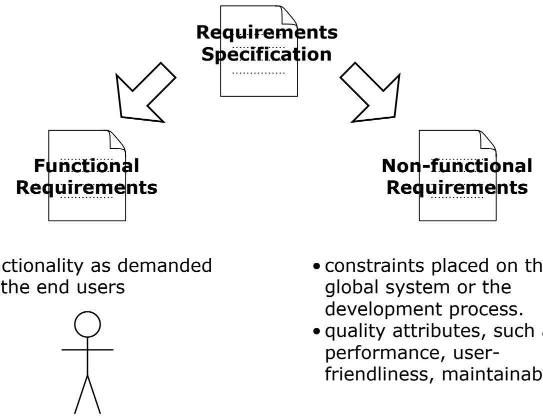
When Architecture?



Designing a software system requires *course-grained decomposition*
⇒ organize work in the development team

Conway’s law
Organizations which design systems are constrained to produce designs which are copies of the communications structure of these organizations. [Conw68a]
- If you have 4 groups working on a compiler; you’ll get a 4-pass compiler

Why Architecture



- functionality as demanded by the end users

- constraints placed on the global system or the development process.
- quality attributes, such as performance, user-friendliness, maintainability, …

ARCHITECTURE
- map requirements onto system structure
  = map function onto form

SCALE ISSUE

Architecture as a Metaphor

Parallels
- Architects are the technical interface between the customer and the contractor.
- A poor architectural design cannot be rescued by good construction technology.
- There are architectural styles or schools.
  + (e.g., “ghotic” in buildings; “client-server” in software)

Differences
- Buildings are tangible, software is intangible.
  ➡ Software Architecture is often expressed via metaphors.
- Buildings are rather static, software is quite flexible.
  ➡ The underlying architecture allows to anticipate changes.
- Building architecture is supposed to be aesthetic.
  ➡ Buildings avoid to mix styles; in software heterogeneity is considered good.
- A building architect caries legal responsibilities.
  ➡ Usually a building architect is not employed by the constructor.

What is Software Architecture?

Software Architecture
- A description of *components* and the *connectors* between them.
  + Typically specified in different views to show the relevant functional and non-functional properties.

Component
- An encapsulated part of a software system with a designated *interface*.
  + Components may be represented as modules (packages), classes, objects or a set of related functions. A component may also be a *subsystem*.

Subsystem
- A component that is a system in its own right, i.e. can operate independently

Connector (a.k.a. Relationships)
- A connection between components.
  + There are *static* connectors that appear directly in source code (e.g., use or import keywords) and *dynamic* connectors that deal with temporal connections (e.g., method invocations).

View
- Represents a partial aspect of a software architecture that shows specific *functional and non-functional properties*.

Functional vs. Non-functional Properties

- See [Bush98a]

Functional property
- Deals with a particular aspect of the system’s functionality. Usually in direct relationship with a particular use case or conceptual class.

Non-functional property
- Denotes a a constraint placed on the global system or the development process. Typically deals with quality attributes that cross-cut the whole system design and are quite intangible.
- Typical non-functional properties
  + Changeability; systems must evolve or perish
  + Interoperability; interaction with other systems
  + Efficiency; use of resources such as computing time, memory, ...
  + Reliability; system will continue to function even in unexpected situations
  + Testability; feasibility to verify that requirements are covered
  + Reusability; ability to reuse parts of software system or process for constructing other systems

Architecture is about tradeoffs

Coupling and Cohesion

Coupling

- Measure of strength for a connector (i.e., how strongly is a component connected with other components via this connector)

Cohesion

- Measure of how well the parts of a component belong together (i.e., how much does the functioning of one part rely on the functioning of the other parts)
    - ➡ Coupling and cohesion are criteria that help us to evaluate architecture tradeoffs.
    - ➡ Minimize coupling and maximize cohesion

However

- The perfect trade-off corresponds to a component that does nothing!
- Coupling at one level becomes cohesion at the next.
    - ➡ More qualitative trade-off analysis is necessary

Patterns

Pattern

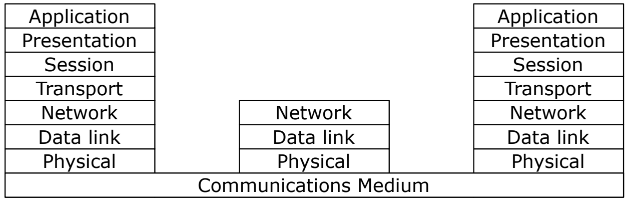
- The essence of a *solution* to a *recurring problem* in a particular context.
    - + Experts recall a similar solved problem and *customize* the solution.
    - + Patterns document *existing* experience.
    - + The context of a pattern states *when (and when not)* to apply the solution.
    - + A pattern lists the *tradeoffs* (a.k.a. forces) involved in applying the solution.

Pattern Form

- Patterns are usually written down following a semi-structured template.
    - + Patterns always have a *name*
    - + Patterns allow experts to have deep design discussions in a few words!

Layered Architecture in Networks

OSI Reference Model



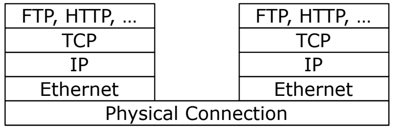
Application
Presentation
Session
Transport
Network
Data link
Physical

Network
Data link
Physical

Application
Presentation
Session
Transport
Network
Data link
Physical

Communications Medium

TCP/IP Stack



FTP, HTTP, …
TCP
IP
Ethernet

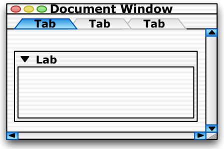
FTP, HTTP, …
TCP
IP
Ethernet

Physical Connection

3-Tiered Architecture

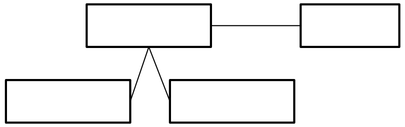
Application Layer

- Models the UI and application logic



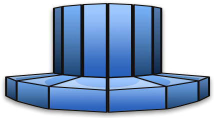
Domain Layer

- Models the problem domain (usually a set of classes)



Database Layer

- Provides data according to a certain database paradigm (usually relational database)



Pattern: Layered Architecture

Context

- Requirements imply various levels of abstraction (low & high level)

Problem

- Need for portability and interoperability between abstraction levels

Solution

- Decompose system into layers;
  each layer encapsulates issues at same level
- Layer n provides services to layer n + 1
- Layer n can only access services at layer n - 1
  - + Call-backs may be used to communicate back to higher layers
  - + Relaxed variant allows access to all lower layers

Tradeoffs

- How stable and precise can you make the interfaces for the layers?
- How independent are the teams developing the different layers?
- How often do you exchange components in one layer?
- How much performance overhead can you afford when crossing layers?

Pipes and Filters Examples

UNIX shells

- `tar cf - .| gzip -cfbest| rsh hcoss dd`

data source = current directory

pipe

filter = compress

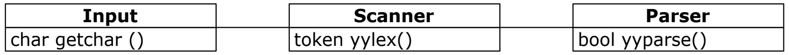
pipe

data sink = remote host

Many CGI-scripts for WWW-forms

- data source is some filled in web-form
- filters are written via a number of scripting languages (perl, python)
- data sink is generated web page
  - + Example: wiki-web pages (http://c2.com/cgi/wiki)

Scanners & Parsers in Compilers



Input
char getchar ()

Scanner
token yylex()

Parser
bool yyparse()

Pattern: Pipes and Filters

Context

- Processing data streams

Problem

- Flexibility (and parallelism) is required

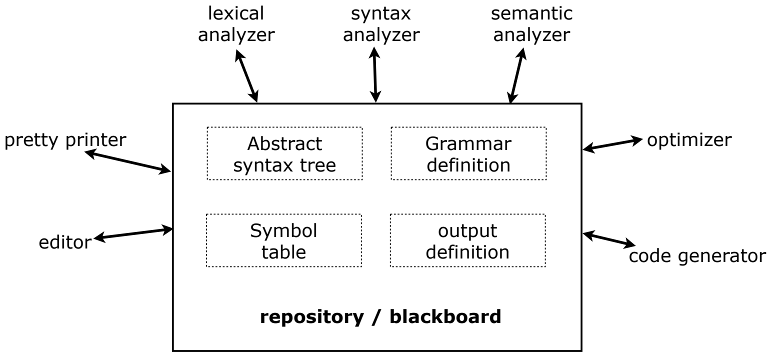
Solution

- Decompose system into filters, each with 1 input- and 1 output stream
- Connect output from one filter to input of another
  - ➡ Need a data source and data sink
- Variants
  - + Push filter: filter triggers *next* one by pushing data on the output
  - + Pull filter: filter triggers *previous* one by pulling data from the input

Tradeoffs

- How often do you change the data processing?
- How well can you decompose data processing into independent filters?
  - + Sharing data other than in/out streams must be avoided
- How much overhead (task switching, data transformation) can you afford?
- How much error-handling is required?

Compilers as Blackboard Architecture



Pattern: Blackboard (a.k.a. Repository)

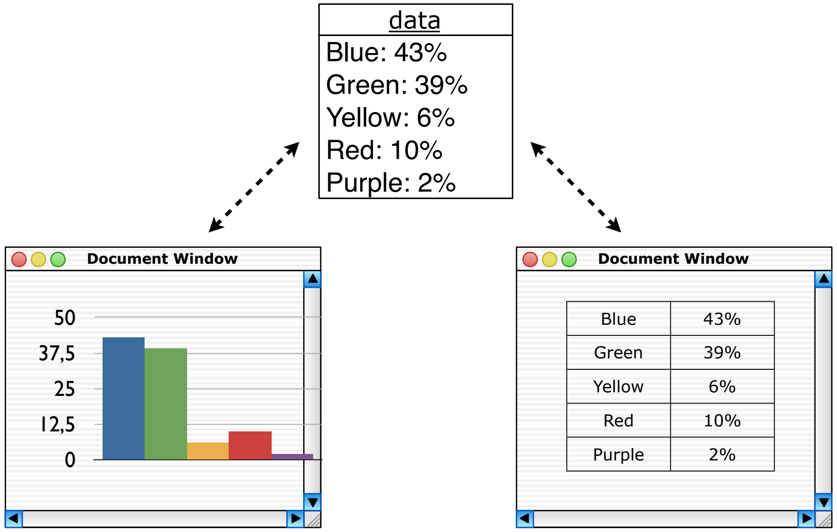
Context
- Open problem domain with various partial solutions

Problem
- Flexible integration of partial solutions

Solution
- Decompose system in 1 blackboard, several knowledge sources and 1 control
  + *Blackboard* is common data structure
  + *Knowledge sources* independently fill and modify the blackboard contents
  + *Control* monitors changes and launches next knowledge sources

Tradeoffs
- How well can you specify the common data structure?
- How many partial solutions exist? How will this evolve?
- How well can you compose an overall solution from the partial solutions?
- Can you afford partial solutions that do not contribute the current task?

Interactive Applications



data
Blue: 43%
Green: 39%
Yellow: 6%
Red: 10%
Purple: 2%

Blue	43%
Green	39%
Yellow	6%
Red	10%
Purple	2%

Pattern: Model-View-Controller

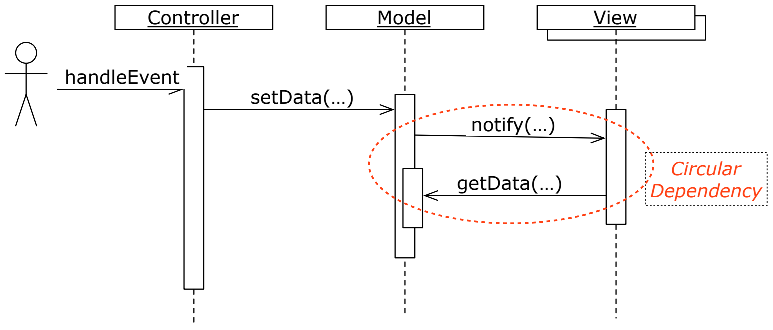
Context
- Interactive application where multiple widgets act on same data

Problem
- Ensure consistency between the various widgets

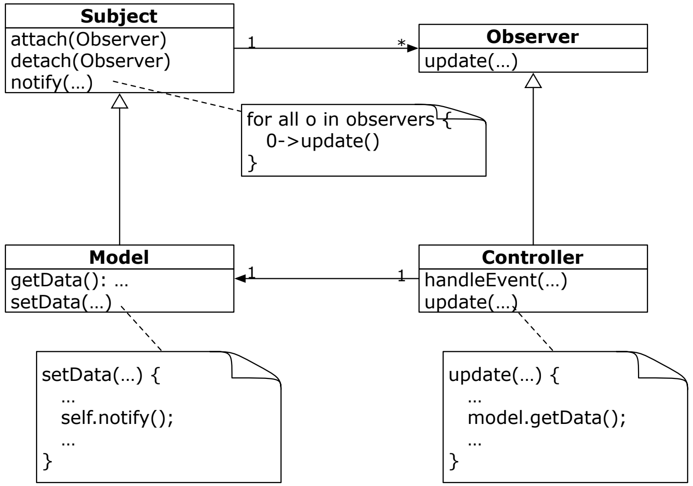
Solution
- Decompose system in a model, and several view-controller pairs
- Model: provides functional core (data)
  + registers dependent views/controllers
  + notifies dependent components about changes (send update)
- View: creates and initializes associated controller + displays information
  + responds to notification events (receive update)
- Controller: accepts user input events + translate events into requests to model and view + responds to notification events (receive update)

Tradeoffs
- How many widgets ? How consistent ? Should they be “plug able” ?
- Increased complexity, especially without library of views/controllers
- Excessive number of updates if not carefully applied
- Close coupling between V-C; average coupling from VC to M

Problem: Circular Dependencies 1-N



Solution: Observer



Subject

attach(Observer)
detach(Observer)
notify(…)

1

*

Observer

update(…)

```
for all o in observers {
   0->update()
}
```

Model

getData(): …
setData(…)

1

1

Controller

handleEvent(…)
update(…)

```
setData(…) {
   …
   self.notify();
   …
}
```

```
update(…) {
   …
   model.getData();
   …
}
```

Pattern: Observer

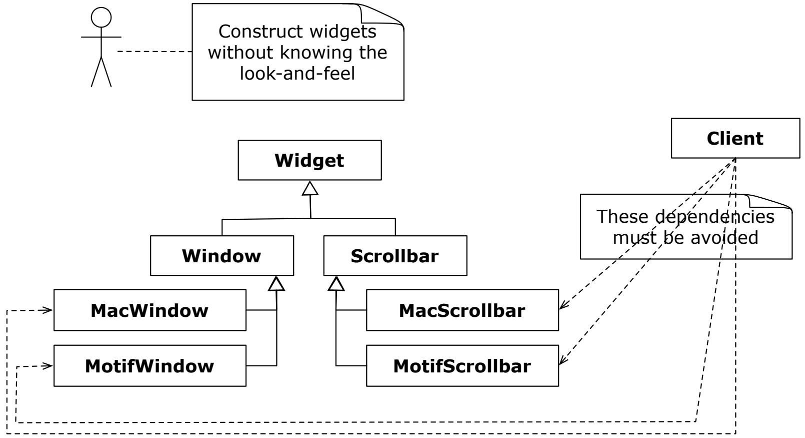
Context
- Change propagation: when one class changes (the subject) others should adapt (the observers)

Problem
- Change propagation implies a circular dependency: (a) adapting requires the observers to access the subject; (b) changing requires the subject to notify the observers

Solution
- Split the circular dependency; move one direction in new superclasses
- Force observers to register themselves on a subject before they will be notified
- Notification becomes anonymous and asymmetrical: subject notifies all observers

Tradeoffs
- Extra complexity: observers will receive more updates than necessary
  + extra program logic to filter the applicable notifications
- Restricts communication between subject and observer

Problem: Constructor Dependencies



Construct widgets without knowing the look-and-feel

Widget

Window

Scrollbar

MacWindow

MotifWindow

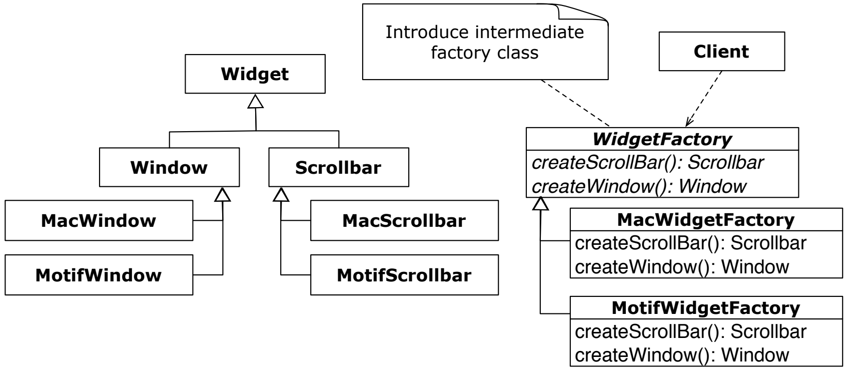
MacScrollbar

MotifScrollbar

Client

These dependencies must be avoided

Solution: Abstract Factory



Introduce intermediate factory class

Client

Widget

Window

Scrollbar

MacWindow

MotifWindow

MacScrollbar

MotifScrollbar

WidgetFactory

createScrollBar(): Scrollbar
createWindow(): Window

MacWidgetFactory

createScrollBar(): Scrollbar
createWindow(): Window

MotifWidgetFactory

createScrollBar(): Scrollbar
createWindow(): Window

Pattern: Abstract Factory

Context

- Class hierarchy with abstract roots representing a family of objects
  + concrete leaves representing particular configurations

Problem

- Invoking constructors implies tight coupling with concrete leaves instead of abstract roots

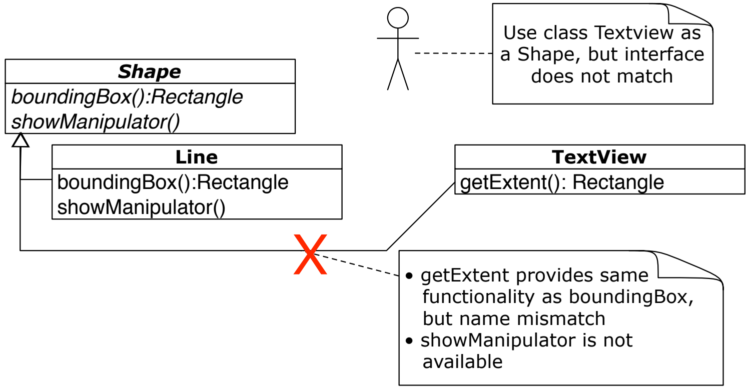
Solution

- Create an abstract factory class with operations for creating all abstract roots
- Create concrete factory classes for all possible configurations.

Tradeoffs

- How many members in the family? How many configurations?
- When do you switch configurations?
- How strict are the configurations?
- Can clients rely on the abstract interfaces?

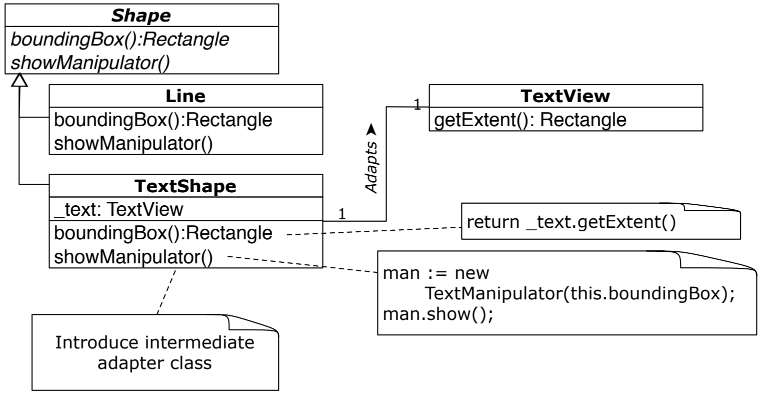
Problem: Interface Mismatch



Use class Textview as a Shape, but interface does not match

- getExtent provides same functionality as boundingBox, but name mismatch
- showManipulator is not available

Solution: Adapter



return _text.getExtent()

```
man := new
      TextManipulator(this.boundingBox);
man.show();
```

Introduce intermediate adapter class

Pattern: Adapter (a.k.a.Wrapper)

Context

- Merge two separately developed class hierarchies

Problem

- Class provides (most of) needed functionality but interface does not match

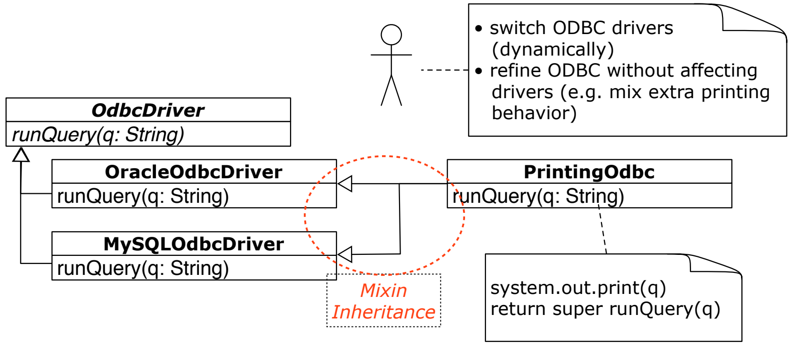
Solution

- Create an adapter class with one attribute of adaptee class
- Adapter class translates required interface into adaptee class

Tradeoffs

- How much adapting is required ?
  + For one class
  + For the whole hierarchy
- How will the separately developed classes evolve ?
- Does the merging work in one direction or in both directions ?
- How much overhead in performance and maintenance can you afford ?

Problem: Alternate Implementation



OdbcDriver
runQuery(q: String)

OracleOdbcDriver
runQuery(q: String)

MySQLOdbcDriver
runQuery(q: String)

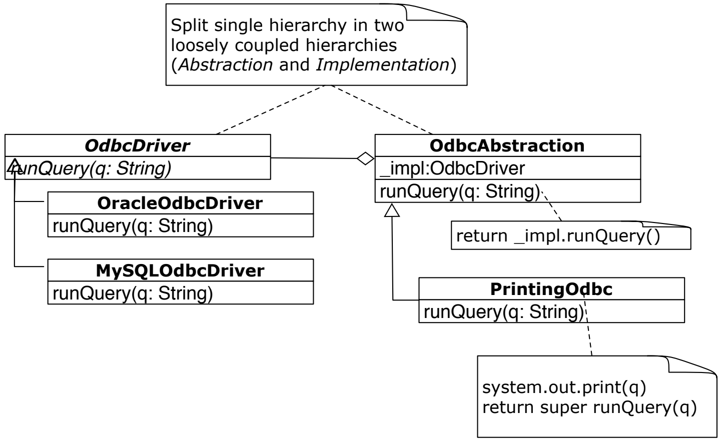
PrintingOdbc
runQuery(q: String)

- switch ODBC drivers (dynamically)
- refine ODBC without affecting drivers (e.g. mix extra printing behavior)

Mixin Inheritance

system.out.print(q)
return super runQuery(q)

Solution: Bridge



Split single hierarchy in two loosely coupled hierarchies (*Abstraction* and *Implementation*)

OdbcDriver
runQuery(q: String)

OracleOdbcDriver
runQuery(q: String)

MySQLOdbcDriver
runQuery(q: String)

OdbcAbstraction
_impl:OdbcDriver
runQuery(q: String)

return _impl.runQuery()

PrintingOdbc
runQuery(q: String)

system.out.print(q)
return super runQuery(q)

Pattern: Bridge

Context

- A class hierarchy represents two perspectives
  - + one a series of implementations
  - + the other a series of variations using these implementations

Problem

- Representing both perspectives requires multiple inheritance
- You cannot dynamically switch implementations

Solution

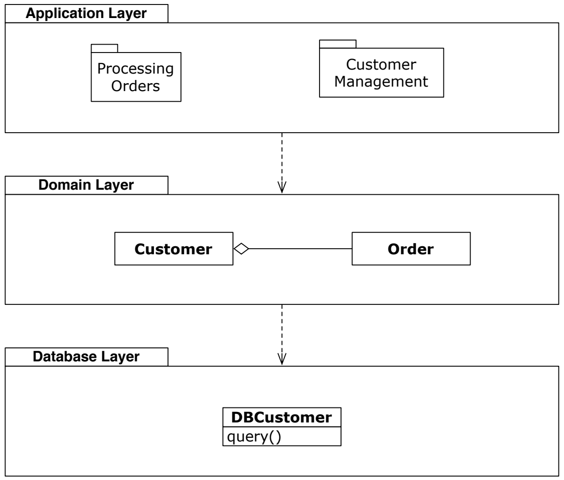
- Split class hierarchy in an *implementation* and an *abstraction* hierarchy
- Introduce an implementation ***bridge*** between them
- Root of abstraction hierarchy represents interface for implementation
  - + Subclasses forward to implementation by invoking on super

Tradeoffs

- Can you clearly separate the implementation from the variation?
- How many implementations and variations exist? Will this increase?
- Do you need to switch implementations dynamically?
- Can you afford the overhead in memory and performance?

UML: Package Diagram

Decompose system in packages (containing any other UML element, incl. packages)



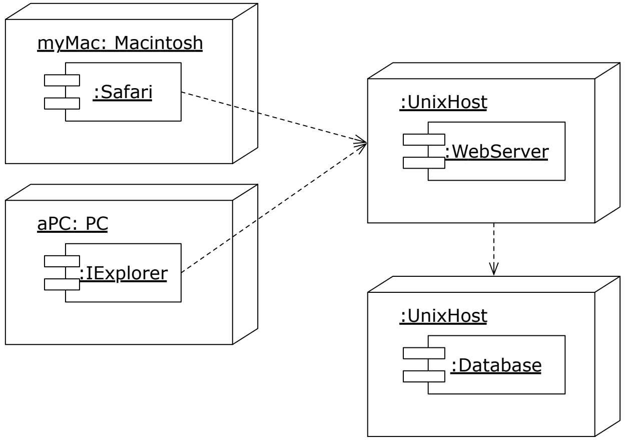
Application Layer
Processing Orders
Customer Management

Domain Layer
Customer
Order

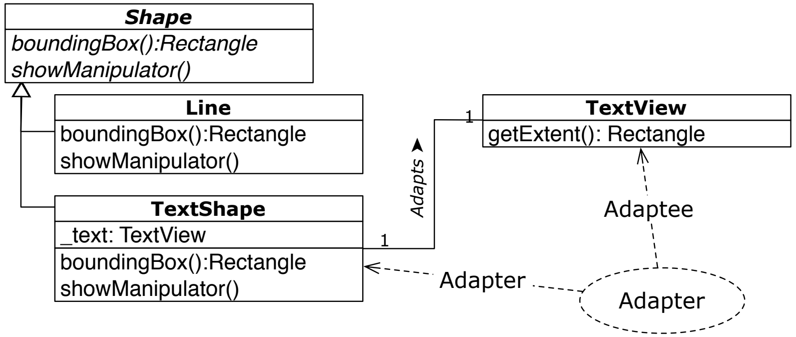
Database Layer
DBCustomer
query()

UML: Deployment Diagram

Shows physical lay-out of run-time components on hardware nodes.



UML: Patterns



Architecture Assessment

Why ?

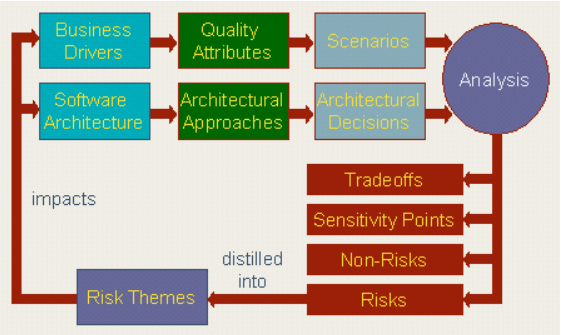
- The earlier you find a problem in a software project, the better.
  - + Identify and assess risks !
- An unsuitable architecture is a recipe for disaster.
  - + A poor architectural design cannot be rescued by good construction technology.
  - + If you wait until the system is built, tackling architectural problems comes at a great cost

Architecture evaluation is a cheap way to avoid disaster.

- Organize review early in the process
  - + An architecture evaluation doesn't tell you "yes" or "no" or "6,75 out of 10".
    - ➡ It tells you were the risks are.

Architecture Tradeoff Analysis Method(ATAM)

- originated from Software Engineering Institute (SEI) at Carnegie Mellon



Answers to two kind of questions:

- Is the architecture *suitable* for the system for which is was designed?
- Which of two or more competing architectures is the most *suitable* one for the system at hand?

ATAM Terminology

Risks are potentially problematic architectural decisions.	The rules for writing business logic modules in the second tier of your three-tier client-server style are not clearly articulated. This could result in replication of functionality, thereby compromising modifiability of the third tier.
Nonrisks are good decisions that rely on assumptions that are frequently implicit in the architecture.	Assuming message arrival rates of once per second, a processing time of less than 30 milliseconds, and the existence of one higher priority process, then a one-second soft deadline seems reasonable.
A **sensitivity point** is a property of one or more components (and/or component relationships) that is critical for achieving a particular quality attribute response.	The average number of person-days of effort it takes to maintain the system might be sensitive to the degree of encapsulation of its communication protocols and file formats.
A **trade-off point** involves two (or more) *conflicting* sensitivity points.	If the processing of a confidential message has a hard real-time latency requirement then the level of encryption could be a trade-off point.

Beware

Patterns

- Patterns define the essence of the solution
  - ➡ misinterpretation is common among people
- Patterns are “Expert” knowledge
  - ➡ “hammer looking for a nail” syndrome
- Patterns introduce complexity (more classes, methods, ...)
  - ➡ cost/benefit analysis

Architecture

- Architecture intends to tackle complexity
  - ➡ say less with more
- Architecture implies tradeoffs
  - ➡ a boxes and arrows diagram is not an architecture (at least consider coupling/cohesion)
- Architectural erosion
  - ➡ law of software entropy
  - ➡ “Big ball of mud” is most often applied in practice

Correctness & Traceability

Correctness

- Are we building the system right ? 🙂
  - + Architecture deals with non functional requirements
    - - Choosing the best architecture involves tradeoffs
  - + Architecture allows to scale up
    - - Organize (testing) work in the team

- Are we building the right system ? 😐
  - + Indifferent

Traceability

- Requirements ⇔ System ? 🙁
  - + Architecture implies extra abstraction level
  - + Software architecture is intangible
    - - Traceability becomes more difficult

Summary (i)

You should know the answers to these questions

- What’s the role of a software architecture ?
- What is a component ? And what’s a connector ?
- What is coupling ? What is cohesion ? What should a good design do with them ?
- What is a pattern ? Why is it useful for describing architecture ?
- Can you name the components in a 3-tiered architecture ? And what about the connectors ?
- Why is a repository better suited for a compiler than pipes and filters ?
- What’s the motivation to introduce an abstract factory ?
- Can you give two reasons not to introduce an Adapter (Wrapper) ?
- Assume the ODBC example after applying the bridge pattern (see Solution: Bridge on page 30). Would it be a good idea for the PrintingOdbc to take advantage of special printing features provided by the Oracle database ? Why ?
- What problem does an abstract factory solve ?
- List three tradeoffs for the Adapter pattern.
- What’s the distinction between a package diagram and a deployment diagram ?
- Define a sensitivity point and a tradeoff point from the ATAM terminology.

You should be able to complete the following tasks

- Take each of the patterns and identify the components and connectors. Then assess the pattern in terms of coupling and cohesion. Compare this assessment with the tradeoffs.

Summary (ii)

Can you answer the following questions ?

- What do architects mean when they say “architecture maps function onto form” ? And what would the inverse “map form into function” mean ?
- How does building architecture relate to software architecture? What’s the impact on the corresponding production processes?
- Why are pipes and filters often applied in CGI-scripts ?
- Why do views and controllers always act in pairs ?
- Explain the sentence “Restricts communication between subject and observer” in the Observer pattern
- Can you compare a bridge with an adapter as a way to build a layered architecture ?
- Can you explain the difference between an architecture and a pattern ?
- Explain the key steps of the ATAM method ?
- How would you organize an architecture assessment in your team ?