CITS4401 Software Requirements and Design
Design Patterns cont'd

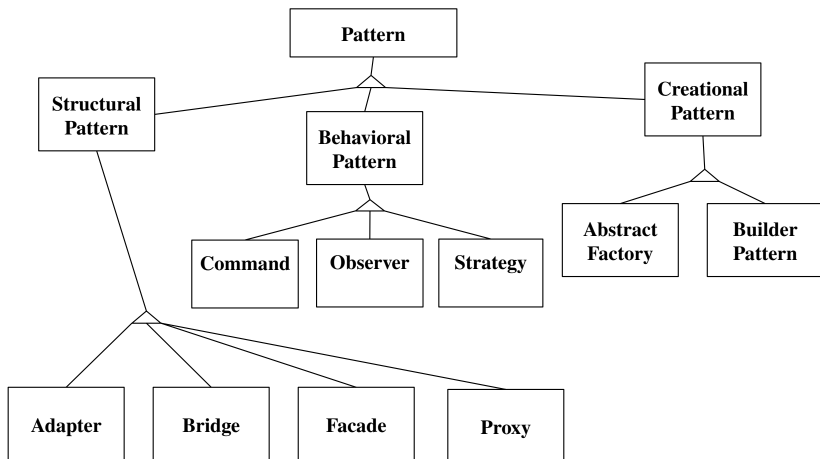
Lecturer: Arran Stewart

Design patterns cont'd

Towards a Pattern Taxonomy

- Structural Patterns
  - Adapters, Bridges, Façades and Proxies are variations on a single theme:
    - They reduce the coupling between two or more classes
    - They introduce an abstract class to enable future extensions
    - Encapsulate complex structures
- Behavioural Patterns
  - Concerned with algorithms and the assignment of responsibilities between objects: Who does what?
  - Characterize complex control flow that is difficult to follow at runtime.
- Creational Patterns
  - Abstract the instantiation process.
  - Make a system independent from the way its objects are created, composed and represented.

A Pattern Taxonomy



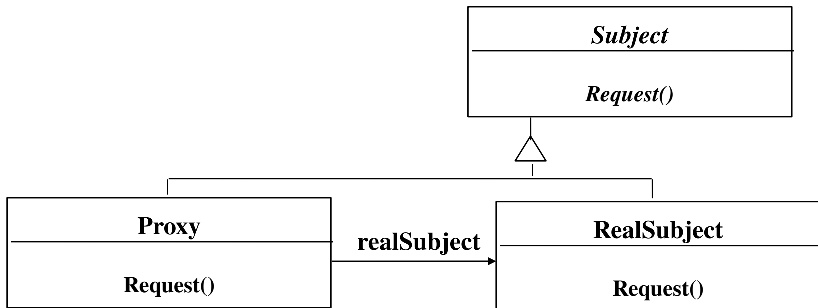
Proxy pattern

Proxy pattern

- What is expensive?
  - Object Creation
  - Object Initialization
- Defer object creation and object initialization to the time that you need the object
- Proxy pattern:
  - Uses another object (“the proxy”) that acts as a stand-in for the real object
  - Reduces the cost of accessing objects
  - The proxy creates the real object only if the user asks for it

Proxy pattern

- Interface inheritance is used to specify the interface shared by Proxy and RealSubject.
- Delegation is used to catch and forward any accesses to the RealSubject (if desired)
- Proxy patterns can be used for lazy evaluation and for remote invocation.

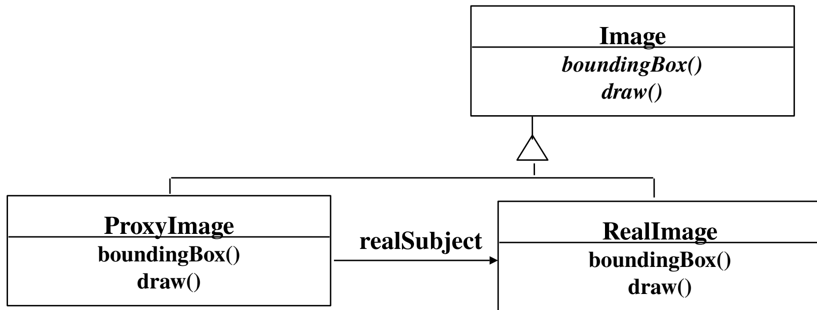


Proxy Applicability

- Remote Proxy
  - Local representative for an object in a different address space
  - Caching of information: Good if information does not change too often
- Virtual Proxy
  - Object is too expensive to create or too expensive to download
  - Proxy is a stand-in
- Protection Proxy
  - Proxy provides access control to the real object
  - Useful when different objects should have different access and viewing rights for the same document.
  - Example: Grade information for a student shared by administrators, teachers and students.

Virtual Proxy example

- Images are stored and loaded separately from text
- If a RealImage is not loaded a ProxyImage displays a grey rectangle in place of the image
- The client cannot tell that it is dealing with a ProxyImage instead of a RealImage
- A proxy pattern can be easily combined with a Bridge



Before

Netscape: Cyberian Outpost - Computers notebooks desktops hardware & software - Buy it Today - Use

Back Forward Reload Home Search Guide Images Print Security Stop

Netsite: http://www.cyberianoutpost.com/

CYBERIAN Outpost

SPECIAL GIFTS FOR DADS & GRADS! CLICK!

OUTPOST TODAY
Jun. 10, 1998

The Cool Place to Shop For Computer Stuff!

We Ship Internationally!
Call: (800)856-9800 | (860)927-2050 | Fax:(860)-927-8375 | E-mail: sales@outpost.com

MAC | PC
DESKTOPS
NOTEBOOKS
PDAS
MEMORY
SOFTWARE
PERIPHERALS
MODEMS
NETWORKING
GAMES
ACCESSORIES
DIGITAL CAMERAS
DOWNLOADS
WHAT'S NEW

56K X2 PC CARD MODEM
(LIMITED TIME ONLY)
PC
$59.95
CLICK TO BUY IT!

SCREAMIN' G3
(LIMITED TIME ONLY)
Mac
$1549.00
CLICK TO BUY IT!

ASTRA 300P FLATBED SCANNER
NEW LOW PRICE!
PC
$69.95
CLICK TO BUY IT!

GLOBAL VILLAGE
56K TelePort
PLUS MAC OS 8

metrowerks

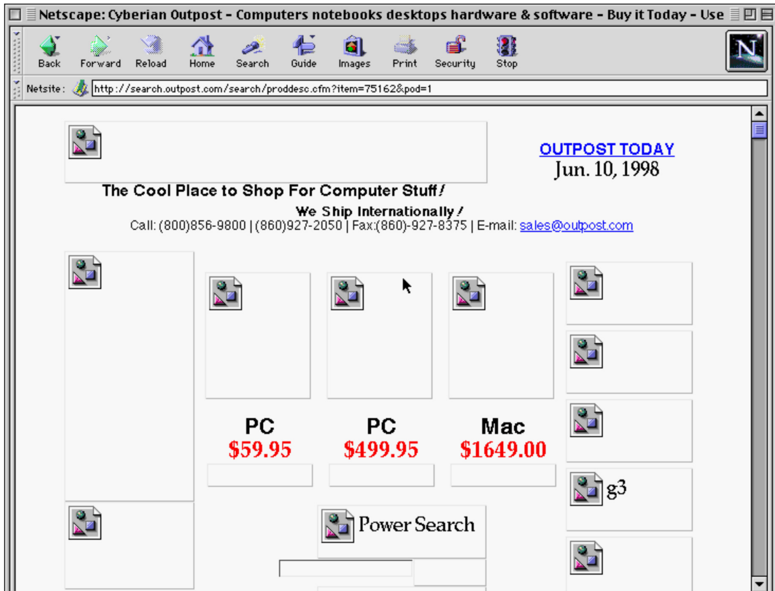
300Mhz/750w/1MB Backside at 150Mhz
MAXpowr G3

WE EAT INTEL PROCESSORS FOR LUNCH!

NEW! BUY IT TODAY - USE IT TODAY!

PRODUCT SEARCH
GO!

After



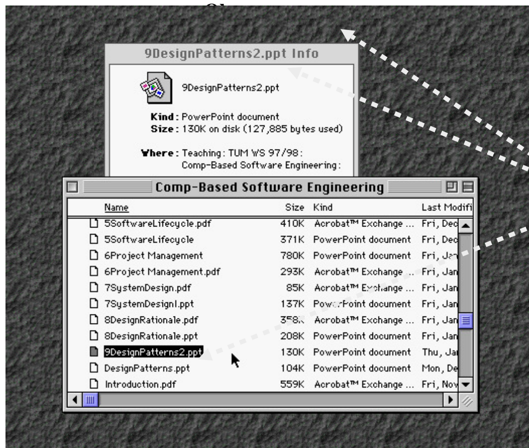
Observer pattern

Observer pattern

- "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."
- Also called "Publish and Subscribe"
- The Observer pattern:
  - Maintains consistency across redundant state
  - Optimizes batch changes to maintain consistency

Observer pattern (continued)

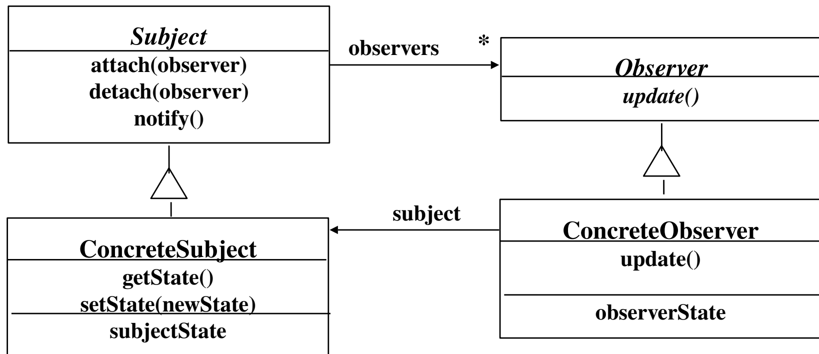
Subject



9DesignPatterns2.ppt

Observer pattern (continued)

- The Subject represents the actual state, the Observers represent different views of the state.
- Subject is a super class (needs to store the observers vector) not an interface.



Sequence diagram

Observer pattern implementation in Java

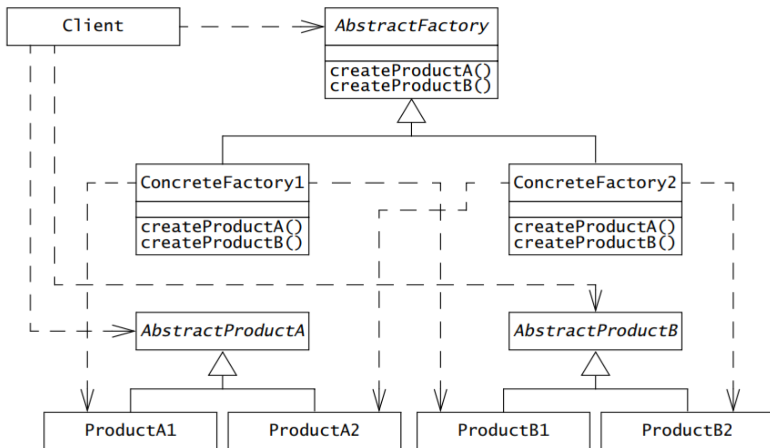
```
// import java.util;
public class Observable extends Object {
    public void addObserver(Observer o);
    public void deleteObserver(Observer o);
    public boolean hasChanged();
    public void notifyObservers();
    public void notifyObservers(Object arg);
}
public interface Observer {
    public abstract void update(Observable o, Object arg);
}
public class Subject extends Observable{
    public void setState(String filename);
  public string getState();
}
```

Abstract Factory pattern

Abstract Factory motivation

- Consider a user interface toolkit that supports multiple looks and feel standards such as Motif, Windows 10 or the finder in MacOS.
  - How can you write a single user interface and make it portable across the different look and feel standards for these window managers?
- Consider a facility management system for a smart house that supports different control systems such as Siemens' Instabus, Johnson & Control Metasys or Zumtobel's proprietary standard.
  - How can you write a single control system that is independent from the manufacturer?

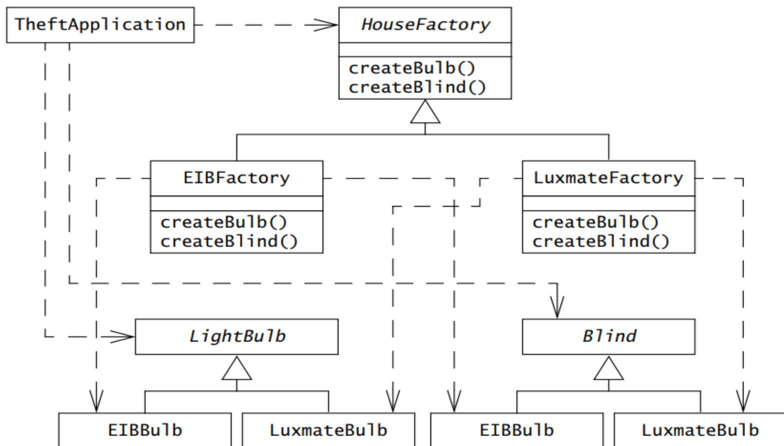
Abstract Factory



Applicability for Abstract Factory pattern

- Independence from initialization or representation:
  - The system should be independent of how its products are created, composed or represented
- Manufacturer independence:
  - A system should be configured with one of multiple family of products
  - You want to provide a class library for a customer ("facility management library"), but you don't want to reveal what particular product you are using.
- Constraints on related products
  - A family of related products is designed to be used together and you need to enforce this constraint
- Cope with upcoming change:
  - You use one particular product family, but you expect that the underlying technology is changing very soon, and new products will appear on the market.

Abstract Factory design pattern to different intelligent house platforms



Other design patterns

- Singleton - used to ensure a class has only one instance and provide a global access point to it. (e.g. static variables of class in java).
- Builder - Separate the construction of a complex object from its representation so the same process can create different representations.
- Flyweight - Use sharing to support large numbers of fine-grained objects efficiently.
- Command - Encapsulate requests as objects, allowing you to treat them uniformly.
- Iterator - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- State - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Summary

- Structural Patterns
  - Focus: How objects are composed to form larger structures
  - Problems solved:
    - To realize new functionality from old functionality,
    - To provide flexibility and extensibility
- Behavioral Patterns
  - Focus: Algorithms and the assignment of responsibilities to objects
  - Problem solved:
    - Overly tight coupling to a particular algorithm
- Creational Patterns
  - Focus: Creation of complex objects
  - Problem solved:
    - Hide how complex objects are created and put together

Summary (2)

Design Pattern	Purpose
Abstract Factory	Encapsulating platform
Adapter	Wrapping around legacy code
Bridge	Allowing for alternate implementation
Command	Encapsulating control flow
Composite	Representing recursive hierarchies
Façade	Encapsulating subsystems
Observer	Decoupling entities from views
Proxy	Encapsulating expensive objects
Strategy	Encapsulating algorithms

Natural language heuristics for selecting design patterns

Phrase	Design Pattern
• "Manufacturer independence" • "Platform independence"	Abstract factory
• "Must comply with existing interface" • "Must reuse existing legacy component"	Adapter
• "Must support future protocols"	Bridge
• "All commands should be undoable" • "All transactions should be logged"	Command
• "Must support aggregate structures" • "Must allow for hierarchies of variable depth and width"	Composite
• "Policy and mechanisms should be decoupled". • "Must allow different algorithms to be interchanged at runtime."	Strategy

Patterns conclusion

- Design patterns
  - Provide solutions to common problems.
  - Lead to extensible models and code.
  - Can be used as is or as examples of interface inheritance and delegation.
  - Apply the same principles to structure and to behavior.
- Design patterns solve all your software engineering problems???
- Reading and studying design patterns will give you a library of solutions, and an awareness of consequences in object oriented software design.

Further reading

- Bruegge & Dutoit, 2010: Chapter 8 and Appendix A
- Gamma, Helm, Johnson, Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1994
- http://www.fluffycat.com/java/patterns.html is a reference and example site for Design Patterns in Java. It contains links to Amazon.com books on design patterns too.
- Design Pattern Card: http://teaching.csse.uwa.edu.au/units/CITS4401/readings/designpatt

Using Rationale to Document Designs

Overview

- Purpose of design documentation
- Good and bad documentation

Purpose of design documentation

Why write design documentation?

- Explain your design choices to later developers
- Ensure they don't duplicate research or work you've already done

Purpose of design documentation

Suppose you need a regular expression library for your project, and the language you are working with (C++, say) doesn't have one built in.

You might spend significant time evaluating alternatives.

Especially if problems occur with the library down the track, it can be very frustrating to later developers if you don't record

- what alternatives were considered?
- why was package X not chosen?
- what would be the consequences for the system of changing the library used?

Purpose of design documentation

- Provide newcomers to the development team with an understanding of the system, its architecture, and design choices made

Sometimes, teams document their methods and classes well – but don't explain how they fit together overall, or what the major components of the system are.

This can make it very difficult for new members of the development team to know how to navigate their way through the codebase, or understand what the major components are.

Purpose of design documentation

- Ensure team has a *shared understanding* of decisions made

Writing down design decisions helps make explicit exactly what they are – otherwise, different team members might have different ideas about what was decided, and why.

Audience

Design documentation records the design choices you made, and reasons for them. It records major choices made about the architecture of the system, and may contain *design models* (for instance, UML diagrams).

It is primarily written for architects, developers and maintainers.

Other sorts of documentation you may encounter, with other audiences:

- **User documentation.** Written for end users and/or system administrators.
- **API documentation.** Written for internal, and possibly external, developers.
- **Cookbooks/examples.** Working code examples, written for developers.

Good and bad design documentation

Design documentation can be good or bad.

Just because there is a *lot* of documentation doesn't mean it is *good*!

Some qualities of *bad* design documentation:

- Hard to navigate
  - hard to find what you need
- Out of date
  - no longer matches up with the current code
- Incomplete/too little detail
  - doesn't explain major decisions
- Too much detail
  - major decisions are hidden in masses of unimportant detail
- Never used
  - consumed time and effort on the part of the people who wrote it, but was never used

Good design documentation

These problems can be overcome by thinking carefully about who will use the design documentation, and how.

Navigation

Hard to navigate

It is very unlikely anyone will read all your design documentation "cover to cover".

They most likely will want to look up an answer to a specific question ("Why didn't we use regular expression library X? It's much faster").

So you need to make sure your design documentation is easy to navigate, and easily searchable.

Navigation

Hard to navigate

Make sure you provide:

- A table of contents
- An index (if in PDF format or hard copy) or search facility
- A *glossary* and/or list of synonyms – especially if the terminology you are using is not widely known, or is ambiguous. (E.g. Terms like “mock”, “wrapper”, “module” or “package” can mean different things to different people.)

Navigation

Hard to navigate

Many teams use facilities like *wikis* to host their design documentation.

Advantages:

- Easy to update
- Easy to search

Disadvantage:

- Can be hard to see the whole documentation "at a glance"
- Hard to tell if search result is still up to date

Current

Out of date

- To be more precise – what is bad is not documentation that is out of date, but the situation where you can't *tell* if it's out of date or not.
- Then you have no idea if it's accurate or not, which is worse than documentation which referred to an older version, but at least was accurate and clearly labelled.

Current

Out of date

Recommendation:

- Label all your design documentation with the version number, and ideally the version control commit ID, that it relates to.
- The documentation can be checked into version control with your code.
- Then if people read it when the software is at version 2.3, and they can see your design documentation relates to version 1.0, they will at least know that it *might* be out of date – it is likely major changes have occurred since then.

Completeness

- Incomplete/too little detail
- Too much detail
- Never used

It's important to ensure your documentation contains *enough* detail, contains the *right amount* of detail, ins't *contradictory*, and will actually be *useful*.

Completeness

- Incomplete/too little detail
- Too much detail
- Never used

Recommendations:

- You don't need to justify absolutely *every* design decision. That will lead to too much detail (and your documentation may never be used).

Completeness

- Incomplete/too little detail
- Too much detail
- Never used

One suggestion (from Pfleeger et al) – consider the following when deciding whether to document the rationale for a decision:

- Was signifcant time spent considering different options?
- Is the design decision critical to achieving a particular requirement?
- Is the decision counterintuitive, or does it raise questions?
- Would the decision be costly to change?

Consistency

- Contradictory

The more documentation you write, and the more your different documents overlap, the more chance there is you may state *contradictory* things in the documentation.

Try to ensure that:

- each document has a clear purpose
- the documents don't overlap with other documents in their purpose or scope
- you only record information that is likely to be needed

Consistency

- Contradictory

Carefully recording what *version* the documentation relates to (as recommended previously) will also help – what appears to be a contradiction might simply be information that differs between versions.

Clarity and writing quality

- Badly written/unclear

Documentation that is badly written or unclear will not be useful.

Many development teams carefully review *code* before it is committed to version control, but do not review their *documentation*.

Both should be done! The documentation should be reviewed for quality and correctness by someone other than the original author.

Clarity and writing quality

- Badly written/unclear

Reviewers should ask themselves:

- Does this document have a clear purpose and scope?
- Is it correct?
- Are design decisions well-justified?
- Are the given explanations clear and succinct?

Rationale overview

- Rationale is the justification of decisions
- Rationale is critical in two areas: it supports
  - decision making and
  - knowledge capture
- Rationale is important when designing or updating (e.g. maintaining) the system and when introducing new staff

Rationale helps deal with change

- Improve maintenance support
  - Provide maintainers with design context
- Improve learning
  - New staff can learn the design by replaying the decisions that produced it
- Improve analysis and design
  - Avoid duplicate evaluation of poor alternatives
  - Make consistent and explicit trade-offs

Rationale activities

- Rationale includes
  - the issues that were addressed,
  - the alternative proposals which were considered,
  - the decisions made for resolution of the issues,
  - the criteria used to guide decisions and
  - the arguments developers went through to reach a decision

Rationale (1)

- Issues
  - Each decision corresponds to an issue that needs to be solved.
  - Issues are usually phrased as questions: How . . . ?
- Proposals / Alternatives
  - Possible solutions that could address the issue under consideration. Includes alternatives that were explored but discarded.

Rationale (2)

- Criteria – Desirable qualities that the selected solution should satisfy. For example,
  - Requirements analysis criteria include usability, number of input errors per day
  - Design criteria include reliability, response time
  - Project management criteria include trade-offs such as timely delivery vs quality

Rationale (3)

- Arguments The discussions which took place in decision making as developers discover issues, try solutions, and argue their relative benefits.
- Resolution The decision taken to resolve an issue. An alternative is selected which satisfies the criteria, supported by arguments for that decision.

Rationale exercise

- Read the excerpt provided in the next slide from the design documents for an accident management system (B & D Chapter 12)
- The excerpt presents the rationale for using a relational database for permanent storage. The argument in presented in prose. Rewrite it in terms of
  - issues, proposals, arguments, criteria and resolutions
- Which version of the document (free prose or issue model) would be easiest to work with during, say, system maintenance? Why?

Rationale exercise (modified from B&D)

One fundamental issue in database design was database engine realization. The initial non-functional requirements on the database subsystem insisted on the use of an object-oriented database for the underlying engine. Other possible options include using a relational database, or a file system. An object-oriented database has the advantages of being able to handle complex data relationships and is fully buzzword compliant. On the other hand, object-oriented databases may be sluggish for large volumes of data or high-frequency accesses. Furthermore, existing products do not integrate well with CORBA, because that protocol does not support specific programming language features such as Java associations. Using a relational database offers a more robust engine with higher performance characteristics and a large pool of experience and tools o draw on. Furthermore, the relational data model integrates nicely with CORBA. On the downside, this model does not easily support complex data relationships. The third option was proposed to handle specific types of data that are written once and read infrequently. This type of data (including sensor readings and control outputs) has few relationships with little complexity and must be archived for extended period of time. The file system option offers an easy archival solution and can handle large amounts of data. Conversely, any code would need to be written from scratch, including serialization of access. We decided to use only a relational database, based on the requirements to use CORBA and in light of the relative simplicity of the relationships between the system's persistent data.

Rationale exercise (modified from B&D)

- Issue: How to realize database engine?
- Proposals:
  - P1: use a Object Oriented database
  - P2: use a relational database
  - P3: use a file system
- Arguments:
  - P1:
    - A+ is able to handle complex data relationship.
    - A+ is fully buzzword compliant.
    - A- may be sluggish for large volumes of data or high-frequency accesses.
    - A- does not integrate well with CORBA.

Rationale exercise (modified from B&D)

- P2: Use a relational DB
  - A+ offers a more robust engine with high performance characteristics.
  - A+ offers a large pool of experience and tools to draw on.
  - A+ integrates well with CORBA.
  - A- does not easily support complex data relationships.
  - P3: Use a file system
    - A+ handles data that are written once and read infrequently (including sensor readings and control outputs which have few relationships).
    - A+ is suitable for data that must be archived for long period of time.
    - A+ can handle large amounts of data.
    - A- needs to write code from scratch.

Rationale exercise (modified from B&D)

- Criteria: Requirement to use CORBA
- Resolution: Use a relational database (proposal 2), based on the criteria and in light of the relative simplicity of the system's persistent data relationships.

Rationale in practice : record and replay

- Facilitator posts an agenda
  - Discussion items are issues
- Participants respond to the agenda
  - Proposed amendments are proposals or additional issues
- Facilitator updates the agenda and facilitates the meeting
  - The scope of each discussion is a single issue tree
- Minute taker records the meeting
  - The minute taker records discussions in terms of issues, proposals, arguments, and criteria.
  - The minute taker records decisions as resolutions and action items.

A Record and replay example: database discussion agenda

- The agenda include 3 issues as the discussion items:
  - I[1] Which policy for retrieving tracks from the database?
  - I[2] Which encoding for representing tracks in transactions?
  - I[3] Which query language for specifying tracks in the database request?

Record and replay example: database discussion

- I[1] Which policy for retrieving tracks from the database?
- Jim: How about we just retrieve the track specified by the query? It is straightforward to implement and we can always revisit it if it is too slow.
- Ann: Prefetching neighboring tracks would not be much difficult and way faster.
- Sam: During route planning, we usually need the neighbor tracks anyway. Queries for route planning are the most common queries.
- Jim: Ok, let's go for the pre-fetch solution. We can revert to the simpler solution if it gets too complicated.

Record and replay example: database discussion minutes

- I[1] Which policy for retrieving tracks from the database?
- P[1.1] Single tracks!
- A- Lower throughput.
- A+ Simpler.
- P[1.2] Tracks + neighbors!
- A+ Overall better performance: during route planning, we need the neighbors anyway.
- {ref: 31/01/2016 routing meeting}
- R[1] Implement P[1.2]. However, the pre-fetch should be implemented in the database layer, allowing use to encapsulate this decision. If all else fails, we will fall back on P[1.1].

Levels of rationale

- No rationale captured
  - Rationale is only present in memos, online communication, developers' memories
- Rationale reconstruction
  - Rationale is documented in a document justifying the final design
- Rationale capture
  - Rationale is documented during design as it is developed
- Rationale integration
  - Rationale drives the design

Agile development and documentation

In 2001, the "Agile Manifesto" was drafted, which captured four major principles for project management, with the goal of developing better software:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- *Individuals and interactions* over processes and tools
- *Working software* over comprehensive documentation
- *Customer collaboration* over contract negotiation
- *Responding to change* over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Agile development and documentation

- Even if you adopt an agile methodology – maintenance will still need to be done!
- And future developers will still need to know why you made the choices you did.

Agile development and documentation

One approach if you are using an agile methodolgy is to do design documentation when software is nearly ready to be handed over to a client.

The assumption is that earlier in the project, the design is likely to still be too much in flux for it to be a good use of time to document it.

Open issues for rationale

- Formalizing knowledge is costly
  - Maintaining a consistent design model is expensive.
  - Capturing and maintaining its rationale is worse.
- The benefits of rationale are not perceived by current developers
  - If the person who does the work is not the one who benefits from it, the work will have lower priority.
  - 40-90% of off-the-shelf software projects are terminated before the product ships.
- Capturing rationale is usually disruptive
- Current approaches do not scale to real problems (e.g., rationale models are large and difficult to search)

Recommended reading

- Bruegge and Dutoit, 2010:
  - Chapter 12
  - §7.4.7 Reviewing System Design