Reengineering Patterns

Reengineering patterns

- Software re-engineering is a relatively new research area
- There is a lack of methodology: How does one approach the problem of re-engineering a software system?
- Reengineering patterns attempt to capture best practices that appear to work well in particular contexts

Reengineering vs. Design Patterns

- Design patterns choose a particular solution to a design problem
- Re-engineering patterns have to do with discovering an existing design, determining what problems it has, and repairing these problems
- Design structure vs. Process of discovery and transformation

Re-engineering patterns

- Artifacts produced by re-engineering patterns can be as concrete as refactored code, or as abstract as insights
- Describe a process that starts with the detection of symptoms and ends with automatic/semi-automatic code refactoring
- Emphasize the context of the symptoms
- Discuss the impact of the changes

Marks of a good RE pattern

- Clarity with which it exposes the advantages, cost, and consequences of target artifacts, with respect to the current system state (not how elegant the result is)
- Description of the re-engineering process: How to get from one state of the system to another

Reengineering pattern form

- Name (usually an action phrase)
- Intent (the essence of the pattern)
- Problem (what makes this problem difficult)
- Solution (might include a recipe of steps)
- Trade-offs (pros & cons of applying the pattern)
- Rationale (why the solution makes sense)
- Known uses (documented instances)
- Related Patterns - What next

Reverse Engineering Patterns

- Setting Direction
- First Contact
- Initial Understanding
- Detailed Model Capture

Setting direction

- Many different factors might affect a re-engineering project
- Technical, ergonomic, economic, and political considerations make it hard to establish and maintain focus
- Hard to set priorities between the many problems of the legacy software
- Danger of focusing on interesting parts rather than what's good for the system

Setting direction patterns

- Agree on Maxims
- Appoint a Navigator
- Speak to the Round Table
- Most Valuable First
- Fix problems, not symptoms
- If it ain’t broke, don’t fix it
- Keep it simple

First contact

- “Where do I start?”
- Legacy systems are large and complex
  - Might need to split it into manageable pieces
- Time is scarce
  - Important to identify the opportunities and risks for the project as soon as possible
- First impressions can be dangerous
  - Always double-check your sources

Chat with the maintainers

- Learn the political and historical context
- Documentation usually records solutions not rationale
- Maintainers will know how the system got to its current state
- System's structure usually reflects the team structure (Conway's Law)

Read all the code in one hour

- Brief but intensive code review with clearly identifiable goal
- Learn coding styles and idioms used
- Browse functional and unit tests
- Look at abstract classes or classes high in the hierarchy
- Singletons represent constant information
- Discover code smells

Skim the Documentation

- Documentation might be outdated or non-existent
- Usually not written with reengineering in mind
- Having a clear goal, you can select the relevant parts fast
- Things to look out for: table of contents, version numbers and dates, figures, screen dumps, formal specs, index

Interview during Demo

- Ask for a demo and interview the person giving it
- This will help find out:
  - Typical usage scenarios
  - Main features offered by the system
  - System components and their responsibilities
  - Anecdotes
- Interview a variety of users: end user, manager, sales person, support personnel, sys-admin, maintainer/developer

Do a Mock Installation

- Check whether all necessary artifacts are available
- Log all failures
- Inability to build might indicate high risk for the reengineering project
- Demands precision about the components required
- Success will increase your credibility

Initial understanding

- Refine ideas from First Contact into an initial understanding
- Document this understanding to support further reengineering efforts
- Allow for iteration and backtracking
- Knowledge must be shared
- Need to communicate, use a language everybody understands

Analyze the persistent data

- Objects kept in a database must be valuable
- However, they might be outdated or of no use anymore
- Data structure in a storage device quite different than when in memory
- Database schema provides a description
- Rough understanding obtained already helps assess which parts of the database are relevant

Speculate about design

- Progressively refine system model by checking design hypotheses against source
- Develop a class diagram of what to expect in the code
- Attempt to match classes in your design to ones in the code
- Adapt class diagram based on mismatches
  - Rename, remodel, extend, seek alternatives

Study the Exceptional Entities

- Use a metrics tool (possibly combined with visualization)
- Study entities with exceptional values
- Interesting issues:
  - Which metrics to collect? Simple is better
  - How to interpret results? Anomalies are not always problematic
  - Important code might have been carefully refactored
  - Difficult to assess the severity of discovered problems

Detailed Model Capture

- Build a detailed model of system parts that are important for reengineering
- Difficulties:
  - Details matter (how to filter out?)
  - Design remains implicit (need to document design decisions that are discovered)
  - Design evolves (important decisions might be reflected in the way the code changes)
  - Studying dynamic behaviour is inevitable

Tie Code and Questions

- Most fundamental and easiest to apply
- Store questions and answers directly in the source, either as comments or language constructs (calls to a global annotator method)
- Difficulties
  - Finding the right granularity
  - Motivating the programmers to write comments
  - Quality of answers
  - Eliminating the annotations

Refactor to understand

- Refactoring can be used to improve the design, but also to help understanding
  - Rename attributes to convey roles
  - Rename methods to convey intent
  - Rename classes to convey purpose
  - Remove duplicated code
  - Replace condition branches by methods
- Regression testing after each change
- Only modify a copy of the code

Step through the execution

- Understand object collaboration by stepping through examples in a debugger
- Collaborations are typically spread throughout the code
- Polymorphism complicates things
- Concrete scenarios cannot be inferred just by reading the source code
- Need representative scenarios
- Does not work that well for time-sensitive, concurrent, or distributed systems

Look for the contracts

- Infer the proper use of class interfaces by studying the way clients use them
- Identify:
  - Proper sequence to invoke methods
  - Valid parameters
  - Export status of methods

Learn from the past

- Study subsequent versions of the system
- Reveals why the system is designed this way
- Configuration management important
- Changes point to important design artifacts
- Repeated growth and refactoring might indicate unstable design
- Some growth and refactoring followed by a stable period indicates mature and stable design

Re-engineering patterns

- Tests: Your Life Insurance!
- Migration Strategies
- Detecting Duplicated Code
- Redistribute Responsibilities
- Transform Conditionals to Polymorphism

Tests: Your Life Insurance!

- Legacy systems often do not have test procedures defined.
- Making changes without introducing bugs is a challenging task
- Certain aspects are difficult to test (concurrency, user interfaces)
- Customers don’t pay for tests but for new features
- An unstable or buggy system is unacceptable
- Hard to motivate programmers to write tests

Write tests to enable evolution

- Properties of well-designed tests
  - Automation (no human intervention)
  - Persistence (document how the system works)
  - Repeatability (can be repeated after each change)
  - Unit testing (tests refer to particular component)
  - Independence (no dependencies between tests)
- An always up-to-date documentation
- Only way to enable software evolution

Grow your test base incrementally

- Balance the costs and benefits of testing by adding tests on an as-needed basis
- Testing everything is impossible
- Previous analysis has identified fragile parts of the system
- Add tests for new features and bug fixes
- Write tests for old bugs (assumes bug history available)

Use a Testing Framework

- Tests are boring to write
- They require considerable test data to be built up and torn down
- Most tests follow the same basic pattern: Create some test data, perform some actions, compare to expected result, clean up data
- Frameworks such as JUnit can be of significant help

Test the interface, not the implementation

- Also known as black-box testing
- Focus on external behaviour rather than implementation details
- Tests survive changes this way
- Exercise boundary values
- Use top-down approach if there are many fine-grained components and not enough time
- Use bottom-up approach if replacing functionality in a very focused part of the legacy system

Record business rules as tests

- Encode business rules explicitly as tests
- Keeps actual business rules, documentation, and implementation in sync
- The rules become explicit
- One needs to record the business rules before reengineering a legacy system
- Enables evolution
- Beware: tests only encode concrete scenarios, not the actual logic of the rules

Write tests to understand

- Record your understanding of a piece of code in the form of executable tests
- Helps validate understanding
- Provides precise specification of certain aspects of the system
- Applies to different levels of understanding
  - Black box: Behaviour
  - White box: Implementation

Migration Strategies

- How to be sure that the new system will be accepted?
- How to migrate while the old system is being used?
- How to evaluate the new system before it is finished?
- Big-bang migration carries a high risk of failure
- Too many changes alienate users
- Constant feedback would be nice but hard to achieve (users are busy)
- Legacy data must survive the migration

Involve the users

- Difficulties:
  - Users can get their job done with the old system
  - People don't want to learn something new unless it really makes a big difference
  - Difficult to evaluate a paper design
  - Hard to get excited about something not ready
- However
  - Users will try new things if their needs are being seriously addressed
  - Users will give you feedback if you give them something useful to use

Build Confidence

- Overcome customer skepticism by demonstrating results at regular intervals
- Both users and developers can measure real progress
- Easier to estimate the cost of smaller steps
- Careful not to alienate original developers
- Management might require bigger demos

Migrate systems incrementally

- Deploy functionality in frequent increments
- Steps:
  - Decompose the legacy system into parts
  - Tackle one part at a time
  - Put tests in place for that part
  - Wrap, reengineer, or replace the legacy component
  - Deploy and obtain feedback
  - Iterate

Prototype the target solutions

- Evaluate the risk of migration by building a prototype
- Identify the biggest technical risks for the reengineering project
  - New system architecture
  - Legacy data migration
  - Performance gains
- Decide on an exploratory (will be thrown away) or an evolutionary (will evolve into the new system) prototype

Always have a running version

- Rebuild the system regularly
  - Have configuration management in place
  - Regression tests must be available
  - Integrate changes as often as possible
- A component does not have to be finished to be integrated
- Some systems might have very large build times. Some re-architecting might be needed

Regression test after every change

- Important for building confidence
- Ensures you always have a running version

Present the right interface

- Wrap a legacy system to export the right abstractions even if they don’t exist in the current implementation

Make a Bridge to the New Town

- Migrate data from a legacy system by running the new system in parallel, with a bridge
- Allows to start using the system without migrating all the data
- The bridge redirects read requests from the new component to the legacy system if the data is not already migrated
- The new component is not aware of the bridge
- The legacy component is adapted to redirect write requests to the new component

Distinguish Public from Published Interfaces

- Published interfaces are interfaces of new components that are not frozen yet
- They are usually available only within a particular subsystem
- Programming languages do not support published interfaces
  - Declare as protected (or package-scope)

Deprecate Obsolete Interfaces

- Give clients time to react to changes to public interfaces by flagging them as obsolete
- Monitor the extent of the use of the deprecated interface, consider removal in a future release
- Language feature in Java
  - @deprecated in Javadoc
  - The compiler issues warnings as well
- Can modify implementation to produce warnings as well

Conserve Familiarity

- Avoid radical changes that may alienate users
- Can be hard (e.g. command-line to GUI)

Use Profiler before Optimizing

- Resist temptation to optimize clearly inefficient code
- A profiler can identify whether there really is a bottleneck
- Optimized code might be unnecessarily complex hindering further reengineering efforts

Detecting duplicated code

- Duplicated code is one of the top code smells
- Good in the short-run, but might have a significant impact in the long run
- 8-12% of industrial software consists of duplicated code
- Hampers the introduction of changes
- Bug fixes have to be applied to all variants
- Scatters the logic of the system

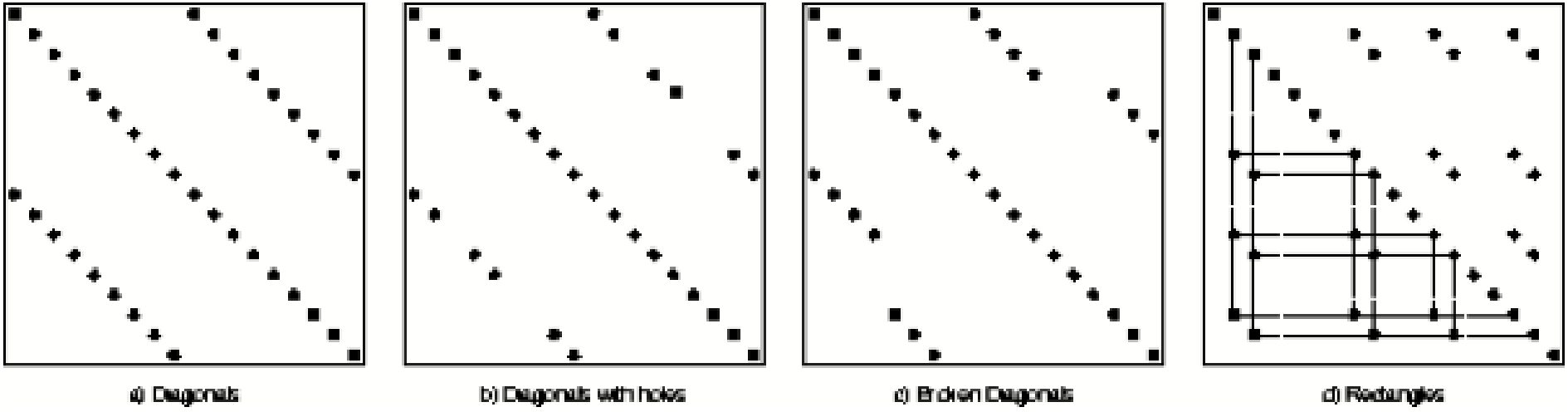
Compare Code Mechanically

- Manual browsing of the code is impractical
- Duplicated code might have modified variable names or slightly different shape
- Steps
  - Normalize by removing comments, tabs, blanks
  - Delete all variables or map them to a common symbol
  - Compare each line with all other lines (hashing might help)
- Rather lightweight approach, easy to compute simple statistics

Visualize Code as Dotplots

- A picture is worth a thousand words
- Visualize the code as a matrix in which the two axes represent two source code files (possibly the same file)
- Dots in the matrix indicate duplication
- Need to normalize beforehand
- Patterns in the dot plot reveal duplication practices

Dotplot examples



1. Exact copies
2. Copies with variations
3. A portion of code has been inserted/deleted
4. Repetitive code elements (e.g. break)

Redistribute responsibilities

- Legacy object-oriented systems may be OO only in name
- Common symptoms:
  - Data containers: Classes containing only data (almost no responsibility)
  - God classes: Classes that implement entire subsystems, commonly just static attributes and methods

Move Behaviour Closer to Data

- Eliminate data containers by moving methods defined in clients to the class that contains the data they operate on
- Need to identify data containers and duplicated client code
- Refactorings such as Extract Method and Move Method can be applied
- Benefits:
  - Data containers become more useful
  - Clients are less sensitive to changes in the data container
  - Code duplication decreases

Eliminate Navigation Code

- Navigation code (`a.b.c.d` or `m1().m2().m3()`) is a sign of misplaced responsibilities and violation of encapsulation
- Such chains of dependency result in changes that have large impact
- Need to push the code from the clients to the suppliers
- Might result in larger interfaces

Split up God Class

- A god class monopolizes control of the application
- Difficult to understand since it contains many abstractions
- Evolution is difficult because most changes affect the god class
- Extract methods and classes out of the god class
- If the god class does not need to be maintained, might be safer to just wrap it

Transform Conditionals to Polymorphism

- Switch statements frequently “smell”
- As a system evolves to handle more cases, conditionals will emerge (quickest way to handle a new case is to add an if statement)
- Makes the code fragile
- Transforming conditionals to polymorphism might be tricky if the inheritance hierarchy is not well developed

Transform Self Type Checks

- Method `m` switches on a private attribute (typically called `type`)
- Move the switch statement to a new method `hook`
- Create a subclass of the current class for each case in the switch statement and move `hook` to all subclasses with the corresponding code
- Make `hook` abstract/deferred in the current class
- No need for `type` anymore

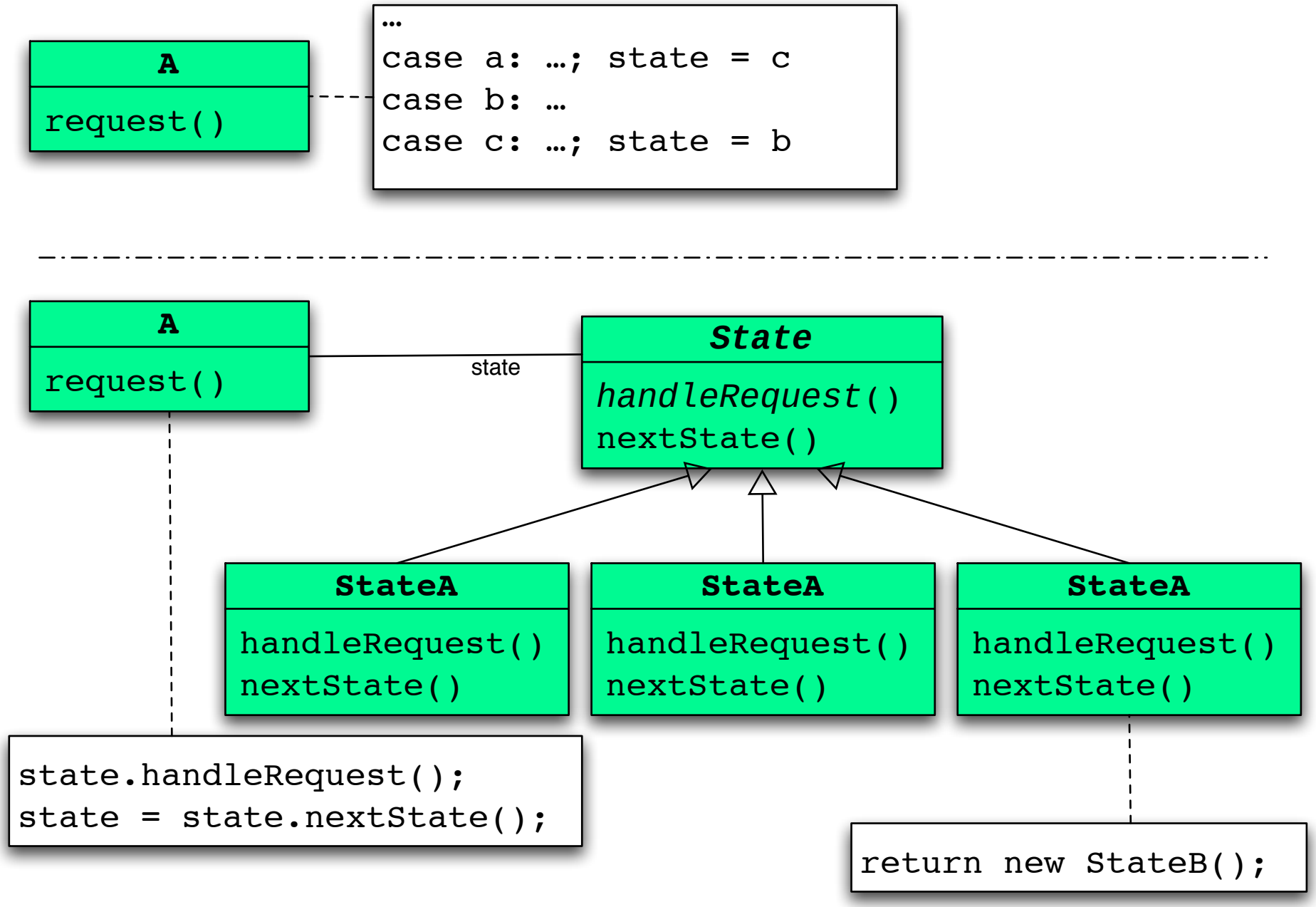
Transform Client Type Checks

- A client switches on the type of the supplier
- Similar problems and solution with Self Type Checks
- Decouples clients from suppliers
- Use of `instanceof` or `getClass` in Java indicates this problem
- If the client switches only on some of the supplier subclasses, a new abstract class might have to be introduced

Factor out State

- Use the State design pattern to eliminate complex conditional code on an object's state
- An object's attributes typically model different abstract states, each with its own behaviour
- Factor the state and the behaviour out into a set of simpler, related classes

Factor out State



Factor out Strategy

- Similar pattern
- Concerned with interchangeable algorithms that are independent of object state
- Improves configurability (new strategies can be plugged in without affecting clients)

Introduce Null Object

- Eliminate conditional code that tests for null
- Create a subclass to act as a null version of the class
- Define default methods in the Null class
- Initialize instances of the class to at least an instance of the Null class
- Remove conditional code

Introduce Null Object