ADVANCED PROGRAMING PRACTICES

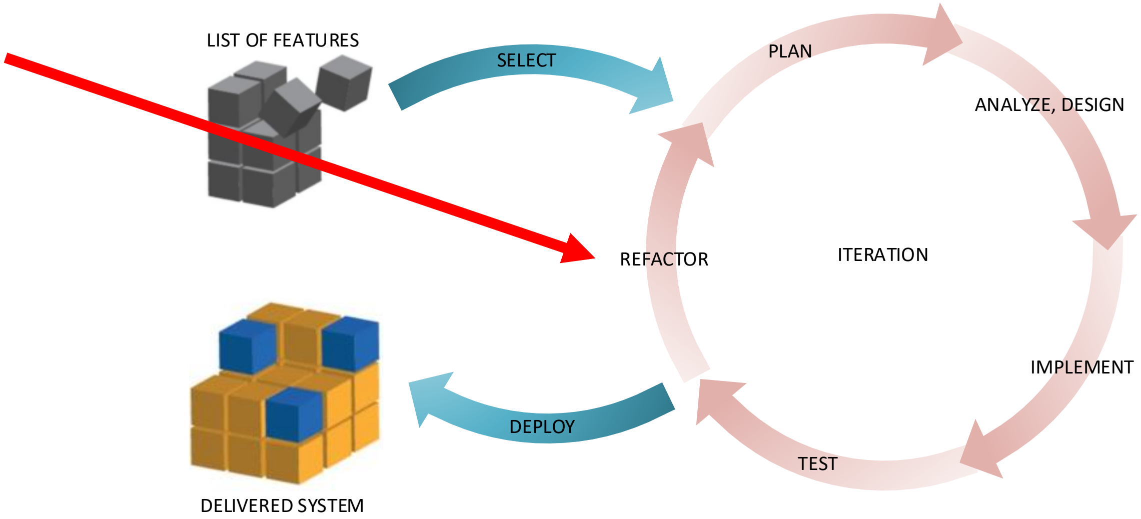
Refactoring

Refactoring: what is it?

- **Definition:** Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its externally observable behavior.
- Refactoring does not fix bugs, but it may help find bugs by scrutinizing code. It may also reduce the further introduction of bugs by cleaning-up code.
- Refactoring does not add new functionality to the system, but it will ease the further adding of new functionality.
- It is an essential part of agile software development such as Extreme Programming or incremental development.

Refactoring: when?

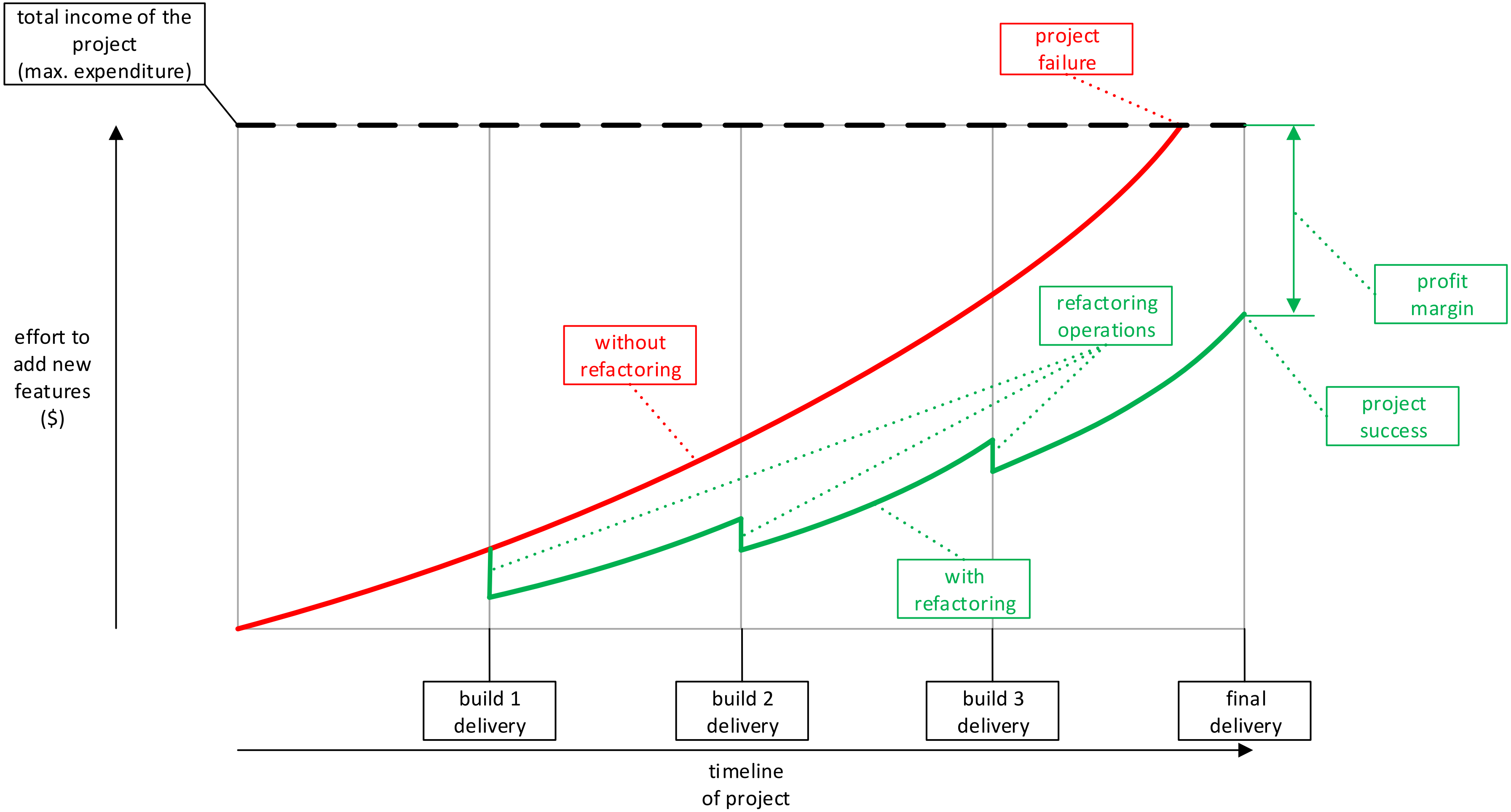
- Constantly during programming
  - Refactoring ought to be done continuously as “bad smells” are encountered during programming.
  - “Bad smells” or “anti-patterns” are portions of design or code that are characterized as potentially confusing and identifies as refactoring targets.
- Between each build in agile software development methods
  - When using iterative or incremental development, a major refactoring stage should precede the beginning of the development of a new build. This will remove slight design problems and ease the addition of further functionality.



Refactoring: why?

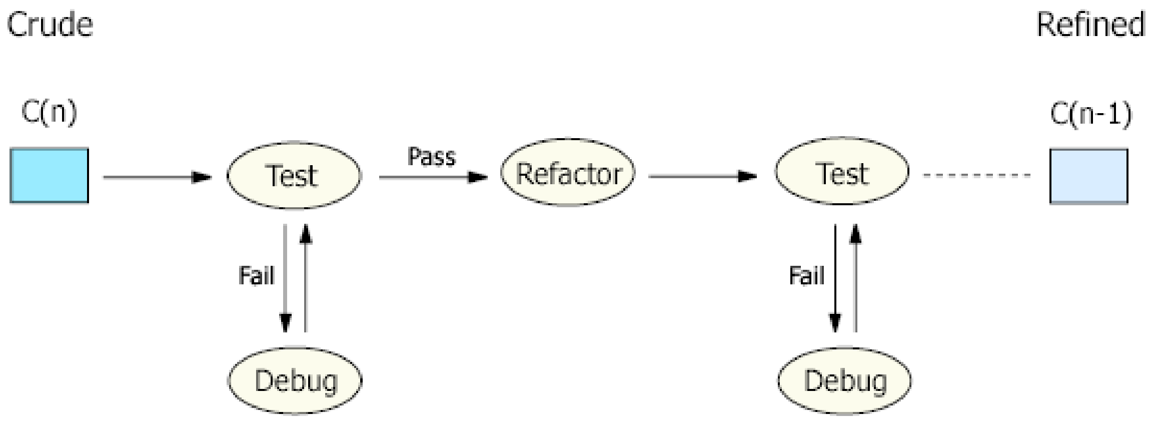
- Refactoring is usually done to:
  - Improve quality
    - improve design quality
    - improve maintainability
    - improve extensibility
    - requires proper testing, so it improves testability
    - helps to find bugs
  - Improve productivity
    - improve code readability & comprehensibility
    - simplify code structure
  - Improve sustainability of development
    - By improving the code’s structural quality, reducing confusion and making the code more understandable, it reduces the effort involved in further development.
    - This is very important in agile software development methods, whose focus on productivity and the high rate of changes is likely to create lower quality code.
    - Without refactoring, agile methods are likely to create code whose further development will be exponentially costly.

Effect of refactoring on project



Refactoring: how?

- Each refactoring is implemented as a small behavior-preserving transformation.
- Behavior-preservation is achieved through pre- and post-transformation testing.
- Refactoring process: **test-refactor-test**



C(x) := Code with x Number of Smells

Refactoring: drawbacks

- **Cost Overhead:** Refactoring is an add-on activity and therefore will incur extra cost in form of time, effort, and resource allocation, especially if elaborated design and code documentation is maintained. However, when done sparingly and only on key issues, its benefits are greater than its overhead. Automated documentation tools, code browsing tools, refactoring tools and testing tools will also diminish the refactoring overhead.

- **Requires Expertise:** Refactoring requires some expertise and experience and considerable effort in going through the process, especially if proper testing is involved. However, this overhead can be minimized by using refactoring tools and automated testing such as with a unit testing framework.

Refactoring patterns

Refactoring: examples

- **Encapsulate Downcast:** A method returns an object that needs to be downcasted by its callers. Refactor by moving the downcast to within the method.

```
Object lastReading() {
      …
      return readings.lastElement();
}
```

```
Reading lastReading() {
      …
      return (Reading) readings.lastElement();
}
```

Refactoring: examples

- **Consolidate Conditional Expression:** You have a sequence of conditional tests with the same result. Refactor by combining them into a single conditional expression and extract it.

```
double disabilityAmount() {
      if (_seniority < 2) return 0;
      if (_monthsDisabled > 12) return 0;
      if (_isPartTime) return 0;
      // compute the disability amount
```

```
double disabilityAmount() {
      if (isNotEligibleForDisability()) return 0;
      // compute the disability amount
```

Refactoring: examples

- **Consolidate Duplicate Conditional Fragments:** The same fragment of code is in all branches of a conditional expression. Refactor by moving it outside of the expression.

```
if (isSpecialDeal()) {
        total = price * 0.95;
        send();
} else {
        total = price * 0.98;
        send();
}
```

```
if (isSpecialDeal())
        total = price * 0.95;
else
        total = price * 0.98;
send();
```

Refactoring: examples

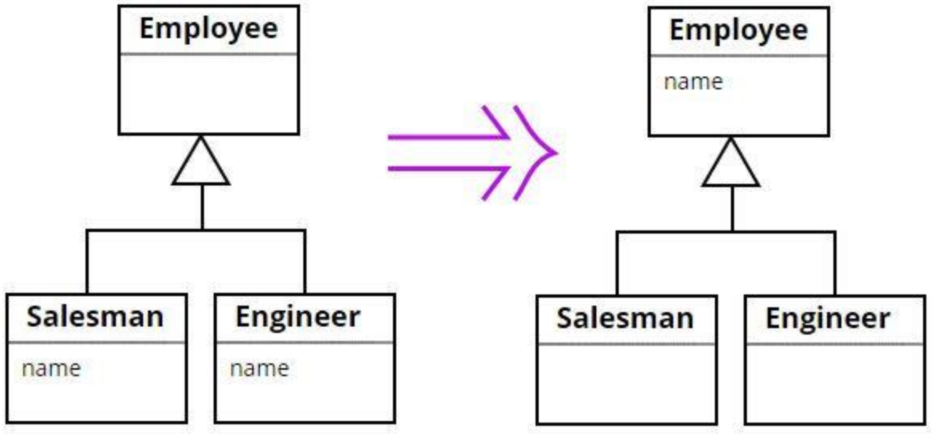
- **Rename Method:** The name of a method does not reveal its purpose. Refactor it by changing the name of the method.

```
int getInvCdtLmt(){
…
}
```

```
int getInvoiceableCreditLimit(){
…
}
```

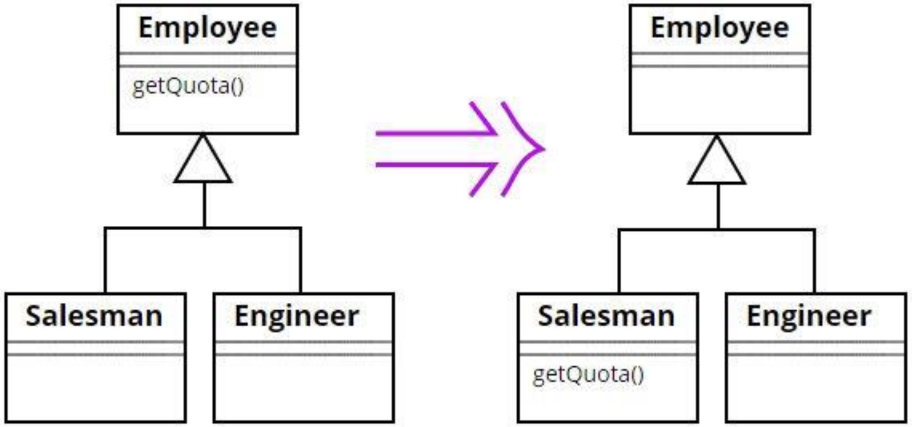
Refactoring: examples

- **Pull Up Field:** Two subclasses have the same field. Refactor it by moving the field to the superclass.



Refactoring: examples

- **Push Down Method:** Behavior on a superclass is relevant only for some of its subclasses. Refactor it by moving it to those subclasses.



Refactoring: practice

- Some refactorings are controversial.
- Some refactorings are arguably not improving code quality.
- Some refactorings can in fact be counter-productive when applied blindly, especially in incremental or iterative development, where design is evolving.
- Have your team adopt a set of refactorings to be applied, and make sure that refactorings are applied in a productive manner.
- Apply in combination with the application of design patterns.
- Use refactoring tools to automate changes, e.g. Eclipse refactoring, and JUnit testing framework.
- For build 2 and 3, you will have to report on the refactoring operations applied between builds.

Refactoring in the project

- A refactoring operation should be done **before** you start working on a new build.
- Establish a list of potential refactoring targets (e.g. 15) using different sources:
  - Code inspections.
  - Discussions among developers.
  - Code review tools.
- Select actual refactoring targets from the list of potential refactoring operations:
  - There may be very numerous potential targets.
  - Select only a few (e.g. 5) actual targets that are likely to have the most positive effects.
  - Tests should be available for all actual refactoring targets.
- For each actual refactoring target:
  - Assess the completeness of the tests that apply to the code being refactored, write more tests if necessary.
  - Run the tests to ensure that the code behaves correctly before the refactoring operation.
  - Determine what transformation you will apply.
  - Apply the refactoring on the code.
  - Run the tests to ensure that the code still behaves correctly after the refactoring operation.

References

- Source Making. Refactoring. http://sourcemaking.com/refactoring
- Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. ISBN-13: 978-0201485677.
- Martin Fowler. Refactoring.com.