Stepwise Feature Introduction

R. J. R. Back
Turku Centre for Computer Science (TUCS) and Åbo Akademi University
Stepwise feature introduction

We consider here a specific method for software construction, where

- the software is constructed in **thin layers**,  
- adding one **feature** at a time to the software, 
- checking that previously added features are **preserved** when adding a new feature, and  
- the layer structure is **refactored** whenever needed.

Refer to this as **stepwise feature introduction**.
Questions

• what kind of software architecture supports this layered approach,
• what are the correctness concerns that need to be addressed,
• what kind of software process supports this structure, and
• how do you manage the complexity of larger systems.
A Metaphore for Layered Software
Complexity of software

- Software systems are very complex: very many components, interacting in different ways

- Software can be described from many viewpoints, in many different dimensions: as code, documentation, diagrams, test runs, use cases, user interfaces etc.

- Software is built in many layers and with deep nesting of components inside components

- Most constructs and concepts are new and unfamiliar to the usual human experiences

- Software is usually constructed in teams, and communication between team members adds to the complexity

We can manage the complexity by using a familiar metaphor (or analogue) that helps us understand the system and to orient ourselves in it.
Some properties of a good metaphor

The metaphor should

- express the basic concepts of software in terms of familiar objects and events in everyday life
- provide a good visual description of the software system
- be reasonably consistent, so manipulations of software correspond to familiar operations in the metaphor
- allow for efficient browsing, checking and editing of software in terms of the metaphor
- should cover the whole life-cycle of software: initial requirements, construction, testing, maintenance, etc
- should scale up to large and complex software.
Our metaphor

The software system starts out as a small village.

- **software components** – houses, buildings
- **identifying components** (names, identifiers) – house names, street addresses, etc
- **communication** between components – by roads, telephone lines, etc
- **servers** – banks, schools, cinemas, etc
- **client** – residential houses in the village, as well as service houses
Life in the village

People in the village use and provide services to each other:

• A person (in a residential house or a service house) can ask a service provider to carry out some task, or to provide some information (so communication is by message passing).

• A person has an address book, so that he or she can locate services.

The village has a history:

• new houses have been built,

• old houses have been torn down,

• the service provided by some houses have been changed.

We assume that each service provider is located in a house of its own.
The village grows

- More people have to be served, better efficiency is needed
- More specialized services are needed and provided
- More and more houses are built, for services and residents

The village becomes a city:

![Diagram of a village becoming a city with various services and buildings.]
Service expansion

- Service houses get two or more floors
- One or more department on each floor
- Each department provides some specialized service
- Floors correspond to layering of the services
- At lower floors are services that have been needed earlier on
- Higher up are more recently created services
- Whenever a new service is needed, a new floor is built, or a new department is added on the top floor.
Layering of services

- A new service may depend on some older service, provided lower down in the building.
- A new service may also replace or extend some old service.
- Contacting department at some floor gives access to the services of that department and of all floors below.
- The top floor provides the most recent and complete service.
The service network grows

A new service may require new services from other service providers:

- The new services may be provided by newly built houses, containing new service providers
- Or, an existing service provider may be extended with the new service required (adding new floors).
- The related new services show up as a layer in the city
A conservative business environment

- A department will not request new services from a service provider, once it has started to make use of services at some layer.
- In particular, a department will not make use of services that have been introduced later on by the service provider.
- The department always communicates with the highest floor of a service provider.
- Hence, the service that the department gets can in fact be redefined by the service provider.
Layering principle

The conservatism implies a layering principle:

each layer can function without the layers that have been built later.
Evolution of the city

The development is evolutionary, with new layers being built on top of existing ones, without changing the old layers.

Sometimes, it becomes too difficult to just simply add a new service feature on top of the existing ones.

One then needs to reconstruct part of a building or maybe even part of the city, so that it becomes easier to add the new service features (software refactoring)

The reconstruction is subject to the following constraint:

The layered structure must be preserved and the layering principle must continue to hold after reconstruction

Otherwise one is free to do any reconstruction one wants.
Software construction method

The metaphor emphasizes

- the **gradual evolution** of a software system towards better and more powerful functionality,

- punctuated with **larger refactorings** when the layering scheme runs into trouble.
Collection of systems

Each layer is always kept fully functional:

- only a subset of the desired features may be implemented by the layer,
- but those that are implemented work correctly,
- the system features provided by a layer can be tested and used independently of the layers higher up.

We are maintaining a collection of systems rather than a single system.
The metaphor emphasizes four main dimensions of software:

- Two dimensions are formed by the city buildings, how they are situated and how they are connected with each other (a view from the top, the city map).

- The third dimension is the layering of the city, corresponding to a view from the side (the city scape).

- The fourth dimension is time, showing how the city has been developed (the city history).
Feature extensions, layers and components (the city architecture)
Features

A software component provides a service.

- The service consists of a collection of (service) features.
- A feature will be implemented as a class.
- This class (usually) extends one or more other classes.
- A class introduces a new feature while at the same time inheriting all the features introduced in the classes that it extends.

We use (multiple) inheritance as the extension mechanism for features.
Example: A text widget

Simple text widget component: display the text, react to mouse clicks, insert new text, delete text, select text, move the insertion point (by clicking the mouse or using the cursor keys.), etc.

Additional features:

saving Save the text in the widget to a file, or replace the text in the widget with the contents of a file.

cut and paste Cut (or copy) a selected piece of text into the clipboard, and paste the contents of the clipboard into the place where the insertion cursor is.

styles Format pieces of text in the widget, e.g. by changing the selected text to boldface, italics, underlined, or colored text.
Extending the text widget

• Add cut and paste to text widget, without compromising earlier features
• Add styles to text widget, without compromising earlier features
• Combine the two extensions in a better text widget.
Managing feature interaction

Simply defining BetterText as an extension of CutAndPaste and of Styles (by multiple inheritance) will not do:

- CutAndPaste will only copy and paste ascii text
- Formatting is then forgotten when copying and pasting in BetterText.

BetterText has to redefine cut, copy and paste operations so that formatting information is also preserved.
Separation of concerns

Layering allows us to separate our concerns:

- The features are added one by one
- Features can be introduced in parallel, working out the mechanism required by each feature separately first (feature introduction)
- The problems arising from the feature interactions can be addressed later, as a separate step (feature combination)
- Feature combination is easier to do when the mechanisms for the features to be combined have been worked out and are open for inspection.
• (a) avoids an arbitrary choice between which feature to introduce first, and allows us to consider the two features in separation.

• (b) and (c) introduce one feature before the other, avoiding the combination step.

• The resulting hierarchy is simpler in (a), but this can happen at the expense of making the second feature more difficult to introduce.
The text widget is extended together with an editor, that displays the text widget and has menus for manipulating it.

There are two layers in this design: the Simple layer and the Better layer.
Layering property

This structure satisfies the layering principle:

- The SimpleEditor together with SimpleText forms a layer that can function without the upper layer.

- The BetterEditor together with BetterText, CutAndPaste and Styles forms an upper layer, that gives the full functionality of the editor.

- The SimpleEditor can call BetterText, but it cannot make use of the new features (cut-and-paste or styles), because it does not know about these.

- We need to build BetterEditor in order to use the new features of BetterText.

- BetterEditor cannot use SimpleText, because the functionality it assumes of the text widget is not implemented in SimpleText.
Correctness Concerns
(a well-designed city)
Design by contract

We assume that the following basic information is given for each class

- a **class invariant** that describes the legal states of an instance of the class
- a **precondition** for each method, describing the states in which the method may be called, and
- a **postcondition** for each method, describing properties of the state that holds after the method has been executed

The postcondition is not strictly necessary, but it may be useful in order to analyze the effect of a method call.
Correctness requirements

We need to check that each class

- is internally consistent,
- respects the other classes that it uses,
- preserves the features of the classes it extends, and
- satisfies its requirements.
**Internal consistency**

The initialization of the class must establish the class invariant.

In addition, each method has to

- preserve the class invariant,
- preserve its loop invariants,
- establish its postcondition, and
- terminate,

whenever the precondition of the method is satisfied initially.
Respect

The class needs to satisfy

- the precondition of each method called from the class, and
- the precondition of each operation performed on values of basic types.
Preserving features

A method in a class that overrides a method of an extended class must behave essentially as the original method.

A customer using the extended class may not notice that it is using the extending class method.

This amounts to requiring that the extending class is a superposition refinement of the original class.

An arbitrary extension of a class is not automatically a superposition refinement, it has to be designed and proved to be one.
Satisfying requirements

The feature has been introduced for a specific reason, i.e., we have certain requirements that need to be realized by the feature.

We therefore also need to show that the feature does in fact satisfy the requirements given for it.

A feature extension may satisfy all the previous correctness criteria, but still not be what we want.
Correctness of software

In carrying out these proof steps, we are allowed to assume that all features on lower layers do satisfy their corresponding requirements.

If we prove that each feature extension satisfies these correctness conditions, then by induction, it will follow that the software system as a whole will satisfy these four correctness conditions.

Note that the final software system will be represented by one feature extension class, for which we then have shown the above properties.
Specifications, Implementations, Views
We may consider BetterText as a specification of a more efficient class (maybe from some class library).

This gives us a layered specification of a software component.
Layered specifications

This gives us a layered specification of a software component.

• BetterText is implemented by EfficientText

• EfficientText is a wrapper for LibraryText

• LibraryText is taken directly from a class library

• SimpleText can be implemented directly by LibraryText
City metaphor

- SimpleText is a house in the village
- SimpleText pretends to provide the required service, but in fact it is just a front end for another service provider, LibraryText, in another city.
- Similarly, BetterText is a front end for another service provider, EfficientText
- EfficientText uses LibraryText for some of its services
- Only BetterText is implemented, not CutAndPaste or Styles.
Correctness concerns

- Need to show that the implementation is a **data refinement** of the specification.

- Requires an abstraction function or relation from the implementation to the specification.

- A data refinement can change the collection of attributes that are used to represent state in a class, a superposition refinement can only add new attributes.

- Usage is monotonic with respect to data refinement: can replace a class by its data refinement without changing the observed behavior of the system.
Using monotonicity

Simplified diagram (BetterEditor uses EfficientText directly):
**BetterText as implementation**

BetterText is an implementation that supports different specifications or **views**:

- **ReadOnlyText** – cannot change text
- **MarkedText** – can highlight (change styles) but not change text
Reasoning with diagrams
**Proof diagrams**

Add extra notation to standard UML class diagrams:

- an exclamation mark in a **class box** means that the class is *internally consistent*,

- an exclamation mark on one end of an **association** means that the class in the other end is *respecting* the usage of the other class, and an exclamation mark next to an **attribute** that the class is respecting the usage of the attribute,

- an exclamation mark on an **inheritance arrow** means that the subclass is a *superposition refinement* of the superclass,

- an exclamation mark on the **implementation arrow** means that the implementing class is a *data refinement* of the original class.

We can say that a class, association, inheritance arrow or implementation arrow in a diagram is **correct**, if we are allowed to place an exclamation mark next to it.
Proof diagrams, cont.

Other markings:

- A question mark next to a diagram entity means that we do not know if the correctness condition holds for this entity.

- No marking means that we do not consider the correctness of this entity.

Establishing a correctness condition can mean different things:

- careful checking, code review
- testing thoroughly
- informal proof of correctness
- formal, mechanized proof
Inferences in proof diagrams 1

Change BetterEditor to use EfficientText directly:

By monotonicity, Better Editor remains internally consistent and respects EfficientText.
Inferences in proof diagrams 2

Assume that we have not proved that BetterEditor is data refined by EfficientText:

Now we do not know if Better Editor remains internally consistent and whether it respects EfficientText.
Implicit arrows in proof diagrams

Some correct arrows can be inferred from the other correct arrows (exclamation marks not shown here):
Requirements and testing
(is this the city we want)
Requirements

Previous correctness criteria guarantee that the system is consistent, but it may still be doing something completely different from what we intended. Must also take system requirements into account.

Requirements are modelled by a customer class. The customer checks that the service it gets is what it expects.

Express service requirement $R$ by

$$R = [P] ; S ; \{Q\}$$

- $P$ and $Q$ are conditions (predicates) on the program variables (object attributes).
- The assume statement $[P]$ means if $P$ then skip else succeed fi.
- $S$ is a statement that describes the requested behavior.
- The assertion statement $\{Q\}$ means if $Q$ then skip else fail fi.
Example customer

An customer class CheckSimpleText for the SimpleText class could be structured as follows:

```
def checkInsert(self, t,s,i):
    [0 ≤ i ≤ |t|]
    server:= self.createServer(t)
    server.insert(i,s)
    s1:= server.get()
    {s1 = t[:i] + s + t[i:]}
```

```
def createServer(t):
    return SimpleText(t)
```
Requirements and unit tests

Requirements are similar to tests, except that they cannot be executed automatically if they have parameters.

A requirement that does not require any parameters can be seen as a test.

Example test method:

```python
def testInsert(self):
    self.checkInsert("abcdefg", "XYZ",2)
```
Requirements for CutAndPaste

Define customer class CheckCutAndPaste, with e.g. following requirement as a method:

```python
def checkCutFollowedByPaste(self,t,i,j,k):
    \[0 \leq i \leq j \leq |t| \land 0 \leq k \leq |t| - j + i\]
    server:= self.createServer(t)
    server.cut(i,j)
    server.paste(k)
    s1:= t[:i]+t[j:]
    s2:= s1[:k]+t[i:j]+s1[k:]
    {server.get() = s2}
```

```python
def createServer(t):
    return CutAndPaste(t)
```
Checking that old requirements still hold

All requirements for SimpleText should also hold for CutAndPaste

We therefore define the class CheckCutAndPaste to be a subclass of CheckSimpleText, so that it inherits all the requirements expressed as methods in CheckSimpleText.

Old methods will be applied to an instance of class CutAndPaste, rather than to an instance of class SimpleText, because we have redefined createServer in CheckCutAndPaste.
Requirements structure

- The customer classes is structured by inheritance in the same way as the server classes.
- No need to indicate inheritance structure explicitly, can use shadows to indicate customer classes.
- Exclamation mark on customer class indicates that the associated requirements are satisfied.
**Metaphore**

Can consider customer classes as the ordinary houses in the village or city.

The inhabitants in these houses are the customers of the services provided by the buildings in the village or city center.

We can have two different customers that make requirements on the same service class, or we can have one customer that make requirements on different service classes (system requirements).

Shadows assume that there is exactly one customer per service class.
Software process
(building the city)
Extreme programming

Extreme programming is a collection of techniques and practices that together form a software process for small team system development (Beck and others). It emphasizes

- short iteration cycles in the development,
- a concentration on producing code rather than documentation,
- avoiding to plan for future extensions,
- frequent testing and integration of software during development,
- frequent refactoring of software structure,
- pair-programming, and
- on-site customers,

among other things.
Stepwise feature introduction process

Extreme programming also emphasizes the incremental construction of software.

The **planning game**:

- the customers list and rank the features they need in the software,
- the programming team estimates the time and other resources needed to implement each feature
- Each iteration cycle is carried out in a fixed time
- Iteration includes only those features that the customer has deemed to be most important and which all can be implemented in the allocated time

Stepwise feature introduction can be seen as a complement to the extreme programming process:

- it describes more precisely how to structure the software and build the components
- the components are developed in the incremental fashion that extreme programming proposes.
Criticism of extreme programming

Extreme programming has also been seen as a new coming of hackerism:

- it de-emphasizes documentation and careful design of the software, and
- it emphasizes the coding part of the software process.

This is not a necessary feature of extreme programming, it is possible to add architecture, documentation and correctness concerns as central parts of the incremental software construction process.
Steps in software construction

The stepwise feature introduction method defines the layering structure of the software, but not the order in which the different components are built (except the obvious one, that they should be built from bottom up).

In particular, we have not said when requirements and tests should be written and when they should be checked.

From the correctness point of view, it does not matter in which order the customer classes and the service classes are written, as long as consistency can be proved for all classes and correctness for all arrows.

Extreme programming prescribes that a tester (unit test class) for a class should be written before the class itself is written. Also, all tests should be executed automatically after each iteration cycle in the software development cycle.

We can generalize the testing approach in extreme programming: requirements should be written before the code.
1. Construct the class CheckSimpleText. It expresses the requirements for a simple text widget.

2. Construct the class SimpleText, using the requirements and tests in CheckSimpleText as design guidelines.

3. Check that SimpleText is internally consistent.

4. Check that CheckSimpleText is internally consistent and

5. Check that CheckSimpleText respect SimpleText.
Constructing the text editor: next layer

Apply steps 1 - 5 to the classes CheckCutAndPaste and CutAndPaste. In addition:

6. Check that the requirements for SimpleText are still satisfied, after the cut-and-paste feature has been introduced.

7. Check that CutAndPaste is a superposition refinement of SimpleText.

Step 7 implies step 6, but requires a proof, whereas step 6 can be partly done by testing.
Experiences in using this method

- I have used the method myself for building an outlining editor in Python.
- We carried out an experiment this summer, combining stepwise feature introduction with extreme programming, as described here. 6 students (3rd year) and 3 Ph.D. students participated in this very successful experiment.
- Presently, an ongoing project in Nokia Mobile Phones tests the method by building a mobile phone application.
- Experiences in using the method for specifying the text widget in Python/Tkinter.
Conclusions

- A method for constructing software by adding one feature at a time, growing the software in thin layers
- A (hopefully) simple metaphor for layered software
- Used UML class diagrams to describe the layered software architecture
- These diagrams can be seen as proof diagrams, that can be used to structure the correctness proofs of large software systems.
- Layering principle gives a collection of fully functional software modules, providing increasingly more functionality
- This allows one to regress to an earlier layer whenever one has problems with an upper layer
- It also allows a branching of software to take place, so that one level is used as a basis for two or more different upper level sequences (software platforms)