Software reuse and generative programming

Promising attacks on essence (from *No Silver Bullet*)

Buy vs. build
Rapid prototyping to aid in the iterative elication of requirements
New metaphor for development: Software systems should be grown rather than built
Key resource is the great designer; these should be identified early and cultivated

Buy vs. build (cont)
Cost of software has always been development cost, not replication cost
– Sharing that cost among even a few clients is a win
– Use of \( n \) copies of a system multiplies the productivity of the programmer by \( n \).
Key issue: Applicability
– Can I use an off-the-shelf product to do my task?
– In 50’s and 60’s, studies showed businesses would not use OTS products because reqts were too specialized
– BUT, that was when dominant cost of computing was in the hardware
– Buyers of $50,000 machines cannot afford to develop custom payroll solutions;
– So they adapt their payroll process instead

Component-based software engineering
Stands to reason that we can buy complete applications, but what about software parts?
Component-based software engineering (CBSE)
– Process model (and lots of supporting technologies) that supports principle of build vs. buy
– Has been pursued for many years in research and in practice
– Very “naïve” form and many sophisticated forms

The “old vision” of CBSE
Assumption: Software-system design should mimic computer-system design
– building a computer is mostly assembling pre-fabricated parts
– parts have standard specifications and can thus be listed in catalogs and ordered to spec
So … we should:
– standardize the technology for representing component interfaces
– wait for industrious software engineers to fill vast repositories with high-quality, well-documented components
– figure out how to index and locate all this great stuff
Obviously, this has not yet happened
Question

Why is this old vision of CBSE naïve?
-or-
What factors, inherent to software, complicate the development and commercialization of vast repositories of reusable software components?

Related questions

What is the definition of component?
I.e., is a component:
- a module (e.g., a class)?
- a procedure?
- a template?
- a collaboration?

One could make a valid case for each of these

More questions

What is the proper granularity of a component?
I.e., should components be:
- small and highly cohesive?
- large and highly functional?
- composed of other components?

Reuse paradox: As components grow larger, reuse payoff increases but reuse likelihood decreases!
- too small and they are not useful enough to be reused
- too large and the set of requirements that they satisfy is not likely to appear in many different projects

Kinds of reuse

Horizontal reuse:
- Components span application domains
- E.g., data structure libraries
Vertical reuse:
- Components come from a specific application domain
- E.g., solar-system kinematics, cell-phone controllers

We will survey two case studies:
- Policy-driven reuse in C++ (horizontal)
- GenVoca and feature-oriented programming (vertical)

Case study 1: Policy-driven reuse in C++

For more information, see:
- A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, Addison Wesley 2001

Problem: Multiplicity of sw design

Software engineering exhibits a rich multiplicity:
- There are many “correct” ways to solve a problem
- An infinity of nuances between “right” and “wrong” variants
Design of a software system is a choice of solutions out of a combinatorial solution space
- Every design decision you make inhibits others, prunes away potentially interesting and useful programs
Challenge for reusable components: Make design decisions that yield robust and useful structures without inadvertently limiting large classes of useful solutions
Policies in C++

Policy classes help in implementing safe, efficient, and highly customizable design elements.

**Policy** defines a class or class template interface:
- design elements perform in conformance with the interface of a policy
- many different implementations of a policy
- thus design elements may be customized by policy implementation

Example: Creator policy

Large source of variation among design solutions concerns the creation of new objects:
- Should new objects be created by allocating new storage off the heap?
- Perhaps new objects should be allocated from a special segment of memory (important if the component is deployed in an embedded system)?
- Perhaps new objects should clone some pre-existing prototype object (important in GUI applications)?

Each variation is a different implementation of a policy for object creation.

We would like to be able to develop components that are neutral wrt the creation policy.

Policy-based component design

Components that are parameterized by the creator policy can be declared as follows:

```cpp
template <class CreationPolicy>
class WidgetManager : public CreationPolicy {
  …
  Widget* w = Create();
  …
};
```

Notice: template parameter used as the base class for WidgetManager
- WidgetManager can be reparameterized by template instantiation
- Effect is to infuse WidgetManager with code in policy implementation

Implementations of Creator policy

```cpp
template <class T>
struct OpNewCreator
{
  static T* Create() { return new T; }
};

template <class T>
struct PrototypeCreator {
  PrototypeCreator(T* proto = 0) : _proto(proto) {
  }
  T* Create() { return _proto ? _proto->Clone() : 0; }
  void SetPrototype(T* proto) {_proto = proto; }
private:
  T* _proto;
};
```

Uses

To use the “new” policy, simply instantiate the template as follows:

```cpp
WidgetManager< OpNewCreator<Widget> > wm;
```

Likewise

```cpp
Widget* fancyWidget;
```

```cpp
WidgetManager< PrototypeCreator<Widget> > wm;
wm.SetPrototype(fancyWidget);
```

Notice: the prototype policy changed the interface of the WidgetManager component!

Use of template template parameters

Often the policy’s template parameter is redundant:
- Gets provided by inference in the definition of the component
- No reason for clients to have to specify it when naming the policy implementation

Solution: Use template template parameters:

```cpp
template<
  template <class Created> class CreationPolicy >
class WidgetManager : public CreationPolicy<Widget> {
  …
};
```

Client code may now specify policy implementation without reference to class `Widget`:

```cpp
WidgetManager<OpNewCreator> wm;
```
Default policy implementations

```cpp
template <template <class> CreationPolicy=OpNewCreator>
class WidgetManager {
public:
CreationPolicy<Widget>  widgetCreator;
CreationPolicy<Foo>      fooCreator;

void manage() {
  Foo*     f = fooCreator.Create();
  Widget*  w = widgetCreator.Create();
}
};
```

Example uses

Use the default policy as follows:

```cpp
WidgetManager<> wm;
```

Use the prototype policy as follows

```cpp
Foo*    fancyFoo;
...
WidgetManager< PrototypeCreator >  wm;
wm.fooCreator.SetPrototype(fancyFoo);
w.
wm.widgetCreator.SetPrototype(fancyWidget);
```

Summary

Policy-based design enables development of components that are highly customizable and thus reusable in a greater number of contexts

“Trick” is to use C++ templates as metaprograms, i.e., little code generators that automatically construct useful variants of a useful component

Metaprogramming (writing programs that write programs) shows up often in reuse contexts

Case study 2: Reusable, hierarchical components

For more information see:


Influence 1: Program families

Classic SE idea: Program families

"We consider a set of programs to constitute a family, whenever it is worthwhile to study programs from this set first by studying their common properties and then determining the special properties of the individual members."

D.L. Parnas [TSE’1976]

Systematic approaches to vertical reuse based on models of program families

What is a “Family”?

Is Netscape a “program” or a family of programs?
How Are Families Developed?

Abstract decisions model of families

Families and OOP

Key idea

GenVoca

Illustrative example
Example continued

Systems are generated by layering plug-compatible components

Example type equations:
- fileSorter = sort[cat]
- viewExtractor = sort[query[sybase]]

Components on left-hand side are generated by applying the refinements in sequence
- Task performed by a GenVoca generator

Example: Amalia family of software analysis tools

Define realm for each representation type
- AST constructor realm for each different notation
- LWA realm for incrementally computing steps
- LTS realm with services for navigating a labeled transition system

Define component (generator) for each translation
- Given a translation: $S \rightarrow T$
  ($S$ = source representation; $T$ = target representation)
  - Add component $\text{trans}[S]$ to realm $T$
- E.g., component that derives steps from ASTs
- E.g., component that derives an LTS from steps

GenVoca model of LTS analysis

3 LTS = \{ inferLTS[LWA], minLTS[LTS] \}
2 LWA = \{ lotosLWA[LOTOS], ltlLWA[LTL] \}
1 LTL = \{ ltlTerm, ltlDNF \}
LOTOS = \{ lotosTerm \}

Sample analyzers

Incremental analyzers:
- lotosLTS = inferLTS[lotosLWA[lotosTerm]]
- ltlLTS = inferLTS[ltlLWA[ltlTerm]]
- ltlLDFLTS = inferLTS[ltlLWA[ltlDNF]]

Minimized analyzers:
- lotosMin = minLTS[lotosLTS]
- ltlLDFMin = minLTS[ltlLDFLTS]

Each analyzer is a component in LTS realm

Case study 3: Enterprise Java Beans

Essence of EJB

Powerful model of application design for client-server enterprise data systems
- design constructs are distributed components
- design and construction is called deployment

Influenced by two ideas:
- Model of separation of concerns based on the deployment of components into containers, which assume responsibilities
- Use of the proxy pattern in distributed object middleware
JavaBeans

Definition: Reusable software component that can be manipulated visually in a builder tool.

Java classes that follow a set of naming and programming conventions:
- Properties
- Events
- Manifest file
- Packaged into a JAR file

Usually implement GUI functionality

Example of a simple bean

class ButtonBean extends ...

implements ...

public String getLabel() { return label; }

void setLabel(String l) { label=l; }

public void addActionListener(ActionListener l)

{...}

public void removeActionListener(ActionListener l)

{...}

"label" property

support for "action" events

What does this enable?

Strict adherence to conventions + Java's reflection capability enables rapid assembly of Bean compositions w/o writing lots of "assembly code".

Each Bean represented graphically as a rectangle listing its properties and events it can issue.

Programmers drag and drop beans from a palette and "wire them together".

A wire between two beans indicates that one receives events from the other.
- Programmer then fills in code that sets property values of recipient based on event data.
- Voila!

BeanBox

Graphical assembly tool

"Container" into which JavaBeans are deployed and configured
- Deployment = dropping a JavaBean into the BeanBox.
- Configuration = creating a wire between two beans.
- Wire creation causes the generation of adapter classes, event handling methods, and configuration code "under the hood".
- Relies on metaprogramming capabilities, specifically use of Java's reflection API.

Value of BeanBox and Java Beans

New model for separating concerns in design.

Applications assembled by deploying components into a container
- Container assumes responsibilities, thereby preventing developer (deployer) from having to think about them.
- E.g., generation of adapter classes, event handlers, configuration code, etc.

Model useful for separating other concerns as well.

Quick background: Proxy pattern

Allows to define a surrogate or place-holder for another object to control access to it:
- Placeholder is called a "proxy".
- Proxy and the subject object must implement common interface.
- Useful for programming with distributed objects:
  - Subject is running inside a server process on some machine on the network.
  - Proxy is a local object that implements operations by remotely executing subject operations on the server.
- Benefit: All of the hassles of distribution are localized in the proxy.
Proxy pattern (diagram)

Distributed proxies

Real subject running inside a server located elsewhere on the network

Proxy code for an operation:
– marshals operation parameters into a request, which can be sent to the server
– blocks waiting for a reply from the server (which will appear as incoming message over the network)
– unmarshals the return data in the reply message
– returns these values to caller

Virtues of distributed proxies

Provided proxy and the subject implement the same interface, application code that wishes to use the subject need not be concerned with distribution issues

Other benefits:
– Subject may be written in a different programming language
– Proxies can incorporate retry policies when request fails
– Etc.

Benefits

Entity beans contain “business logic” without concern for session management, persistence, etc.

Beans deployed in reusable application servers (e.g., JBoss)

Result: Enables the rapid construction of open and accessible enterprise information systems

Back to EJB

Key idea: Distributed objects + local components = distributed components

An Enterprise Java Bean is a distributed component that is deployed in a container running on an application server
– Entity beans represent data records
– Session beans represent “persistent state” such as a shopping cart

EJB container responsible for:
– Managing issues of data persistence, session persistence, transactions, etc